ISBN 978-1-939133-19-9

Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20) November 4–6, 2020

conference

proceedings

14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)

November 4–6, 2020

Sponsored by

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

OSDI '20 Sponsors

Silver Sponsors











Bronze Sponsors




Industry Partners and Media Sponsor

ACM *Queue*
DMTF
FreeBSD Foundation

USENIX Supporters

USENIX Patrons

Bloomberg • Facebook • Google
Microsoft • NetApp

USENIX Benefactors

Amazon • AuriStor • Oracle
Thinkst Canary • Two Sigma • VMware

USENIX Partners

Top10VPN

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)

November 4–6 2020

© 2020 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-19-9

Cover Image created by freevector.com and distributed under the Creative Commons Attribution-ShareAlike 4.0 license (https://creativecommons.org/licenses/by-sa/4.0/).

Symposium Organizers

Program Co-Chairs
Jon Howell, *VMware Research*
Shan Lu, *University of Chicago*

Program Committee
Rachit Agarwal, *Cornell University*
Lorenzo Alvisi, *Cornell University*
Tom Anderson, *University of Washington*
Sebastian Angel, *University of Pennsylvania*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Andrew Baumann, *Microsoft Research*
Irina Calciu, *VMware Research*
George Candea, *EPFL*
Rong Chen, *Shanghai Jiao Tong University*
Wenguang Chen, *Tsinghua University*
Vijay Chidambaram, *The University of Texas at Austin and VMware Research*
Byung-Gon Chun, *Seoul National University*
Allen Clement
Natacha Crooks, *University of California, Berkeley*
Dilma Da Silva, *Texas A&M University*
Alexandra Fedorova, *University of British Columbia*
Jason Flinn, *Facebook*
Roxana Geambasu, *Columbia University*
Yossi Gilad, *The Hebrew University of Jerusalem*
Haryadi Gunawi, *University of Chicago*
Andreas Haeberlen, *University of Pennsylvania*
Tim Harris, *Amazon*
Chris Hawblitzel, *Microsoft Research*
Gernot Heiser, *University of New South Wales and CSIRO's Data61*
Y. Charlie Hu, *Purdue University*
Ryan Huang, *Johns Hopkins University*
Rebecca Isaacs, *Twitter*
Frans Kaashoek, *Massachusetts Institute of Technology*
Manos Kapritsos, *University of Michigan*
Baris Kasikci, *University of Michigan*
Kimberly Keeton
Anne-Marie Kermarrec, *EPFL*
Ana Klimovic, *Google Research and ETH Zurich*
Jinyang Li, *New York University*
Wyatt Lloyd, *Princeton University*
Jay Lorch, *Microsoft Research*
Xiaosong Ma, *Quatar Computing Research Institute*
Kathryn S. McKinley, *Google*
James Mickens, *Harvard University*
Robert Morris, *Massachusetts Institute of Technology*
Derek Murray, *Google*
Madan Musuvathi, *Microsoft Research*
Bryan Parno, *Carnegie Mellon University*
Simon Peter, *The University of Texas at Austin*
Don Porter, *The University of North Carolina at Chapel Hill*
Dan Ports, *Microsoft Research*
Costin Raiciu, *University Politehnica of Bucharest*
Malte Schwarzkopf, *Brown University*
Ryan Stutsman, *University of Utah*
Michael Swift, *University of Wisconsin—Madison*
Kaushik Veeraraghavan, *Facebook*
Rashmi Vinayak, *Carnegie Mellon University*
Xi Wang, *University of Washington*
Yang Wang, *The Ohio State University*
John Wilkes, *Google*
Emmett Witchel, *The University of Texas at Austin*
Harry Xu, *University of California, Los Angeles*
Tianyin Xu, *University of Illinois at Urbana–Champaign*
Junfeng Yang, *Columbia University*
Ding Yuan, *University of Toronto*
Nickolai Zeldovich, *Massachusetts Institute of Technology*
Irene Zhang, *Microsoft Research*
Yiying Zhang, *University of California, San Diego*
Lidong Zhou, *Microsoft Research*
Yuanyuan Zhou, *University of California, San Diego*

Steering Committee
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Jason Flinn, *Facebook*
Casey Henderson, *USENIX Association*
Kimberly Keeton
Hank Levy, *University of Washington*
James Mickens, *Harvard University*
Brian Noble, *University of Michigan*
Timothy Roscoe, *ETH Zurich*
Margo Seltzer, *University of British Columbia*
Geoff Voelker, *University of California, San Diego*

External Reviewers

Joy Arulraj
Mahesh Balakrishnan
Fred Chong
Peter Chubb
Zhihao Jia
Gerwin Klein
Jing Li
Ashlie Martinez
Justin Meza
Rajesh Nishtala
Rohan Padhye
Mark Silberstein
Adriana Szekeres
Chunqiang Tang
Carl Waldspurger
Ben Y. Zhao

Message from the OSDI '20 Program Co-Chairs

Dear colleagues,

Welcome to the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)!

This year's program offers an unprecedented 70 exceptional papers. These papers represent the many strengths of our community and cover a wide range of topics, including file and storage systems, networking, scheduling, security, formal verification of systems, cluster management, system support for machine learning, hardware, consistency, consensus protocols, debugging, and, of course, operating systems design and implementation.

Our committee received a bumper crop of 400 submissions, an increase of more than 50% over OSDI '18. This growth demanded that we modify the review process and grow the committee at the last minute to handle the load. When you bump into a PC member, give them a huge thank you! Sixty-five members participated, including academics, industrial researchers, and industrial practitioners. Papers received two reviews in the first round; 305 advanced to round two, where they received an additional review. Of those, 177 advanced to round three, where they received three more reviews. For a small number of papers, where opinions were divided or where a paper was particularly specialized, we solicited additional expert reviews. In total, the PC and external reviewers wrote more than 1.6 million words in more than 1,600 thoughtful reviews.

After a rigorous online discussion across the full PC, the heavy PC members discussed 101 papers in a virtual 2-day PC meeting. The PC chairs strove to ensure that all the discussed papers received full and fair consideration, coming to a consensus agreement in almost every case. Papers were placed into high-level categories according to their main topic so that similar papers could be discussed together at the PC meeting. All discussed papers received a summary of the PC discussion written by a heavy PC member. In the end, the PC selected 70 papers for presentation at the conference, resulting in an 18% acceptance rate, similar to prior years. Each of the accepted papers was allocated an additional two pages and shepherded by a member of the heavy PC to help the authors address the reviewers' comments in their camera-ready versions.

After finalizing the program, we created a separate committee to decide the Jay Lepreau Best Paper Awards composed of PC members with no conflicts with the papers under consideration. PC members nominated papers for these awards. We selected four papers with at least two nominations for best paper as candidates for the award. After reading the nominated papers and considering the reviews from the full PC, the awards committee agreed on three Jay Lepreau Best Paper Awards.

As PC co-chairs, we stand on the shoulders of so many who did a tremendous amount of hard work to make OSDI '20 a success. First, we thank the authors of all submitted papers for choosing to send their work to OSDI. Thanks also to the program committee for their hard work in reviewing and discussing the submissions and in shepherding the accepted papers. We thank Vijay Chidambaram and James Mickens for organizing the Ask Me Anything sessions, and we thank Malte Schwarzkopf, Aastha Mehta, Natacha Crooks, and Brian Noble for organizing the student mentoring sessions. We are delighted that Anjo Vahldiek-Oberwagner, Eric Eide, and Ryan Stutsman have organized the artifact evaluation process. We are also grateful to the external reviewers who provided additional perspectives. We thank the USENIX staff, who have been fundamental in organizing OSDI '20 in an especially difficult year. Finally, OSDI wouldn't be what it is without our attendees—thank you for listening to our speakers, asking challenging and insightful questions, sharing your ideas with others, and networking with one another in Slack!

We hope you will find OSDI '20 interesting, educational, and inspiring!

Shan Lu, *University of Chicago*
Jon Howell, *VMware*
OSDI '20 Program Co-Chairs

Message from the OSDI '20 Artifact Evaluation Committee Co-Chairs

It is our pleasure to report on the artifact evaluation process conducted as part of OSDI '20. This year's conference represents the first time that OSDI has included an artifact evaluation committee (AEC), and it immediately follows the inaugural year for artifact evaluation at SOSP.

The goal of artifact evaluation is to incentivize authors to invest in the broader scientific community by producing artifacts that illustrate their claims, enable others to validate those claims, and accelerate future scientific progress. A paper with artifacts that have passed the artifact evaluation process is recognized in two ways: first by badges that appear on the paper's first page, and second by an appendix that details the artifacts.

Process

In designing the artifact evaluation process for OSDI, we aimed to bridge the processes from earlier USENIX conferences (USENIX Security) and the prior effort from ACM SOSP. USENIX previously used a single-badge process, whereas SOSP used a system based on the ACM's artifact review and badging policy. After deliberation, we decided on a three-badge approach to evaluation. This helps establish congruence between the processes for SOSP and OSDI, and the finer granularity of a multi-badge system encourages participation even when full artifacts cannot be shared or specific results are too challenging for the committee to reproduce. The three badges that we used for OSDI are:

- **Artifacts Available:** Have the artifacts associated with the paper been made available for retrieval both permanently and publicly?
- **Artifacts Functional:** Do the artifacts conform to the expectations set by the paper in terms of functionality, usability, and relevance?
- **Results Reproduced:** Can the AEC use the submitted artifacts to obtain the main results presented in the paper?

The criteria for each badge are independent; for example, an artifact does not need to be deemed available or functional in order to be considered for the "Results Reproduced" badge. The third badge corresponds to the "Results Replicated" badge at SOSP '19 but differs in name. The OSDI badge name matches terminology recommended by the National Information Standards Organization (NISO).

Evaluation

To form the artifact evaluation committee, we issued an open invitation to the systems community for self-nominations. From the self-nominations, we selected 40 early-career researchers and graduate students based on their levels of expertise.

After the decisions for OSDI '20 paper submissions were distributed, the authors of accepted papers were invited to submit artifacts for evaluation. (Thus, the artifact evaluation process had no effect on which papers were chosen to appear at OSDI.) Authors had one and a half weeks, until August 28, to respond to the call for artifacts. At artifact-submission time, authors were required to choose the badges for which their submission would be considered. The overwhelming majority of submissions applied for all three badges. Each artifact was accompanied by the accepted version of its associated paper so that the AEC could evaluate each artifact against its paper's claims.

A total of 49 artifacts were submitted for evaluation. The AEC members bid on artifacts, and we assigned two or three reviewers for each submission—three if the submission applied for the "Results Reproduced" badge, and two otherwise. After bidding, the AEC had five weeks, until October 9, to make judgments.

Evaluation started with an attempt to build the artifact (where appropriate). Next, AEC members tried to repeat some or all of the experiments described in the artifact's paper. AEC members were cognizant that it would be difficult to reproduce certain reported results, e.g., due to environmental or time limits. Reviewers were able to communicate with authors and regularly did so for clarifications and for help in debugging issues, with HotCRP preserving single-blind reviewing. Along the way, AEC members assessed each artifact's completeness, documentation, and apparent ease of reuse. After all reviews were submitted, the AEC held an online discussion to decide if—for each artifact—it met, exceeded, or fell below the expectations set by its paper.

Overall, the process generated 133 reviews and 1,180 comments with an average of about 3,000 words of combined review text and comments per artifact.

Results

OSDI '20 accepted 70 papers; in comparison, SOSP '19 accepted 38. Correspondingly, we received a greater number of submitted artifacts: 49 versus 23. We also saw an increase in the fraction of papers that chose to participate: 70%, up from 61% at SOSP '19. We hope that this trend will continue as artifact evaluation becomes a regular part of our community's conferences.

Of the 49 submitted artifacts, the AEC found that 48 met or exceeded expectations for at least one of the three badges. Per the choices of the authors, not all artifacts were considered for all badges.

- 47 artifacts received the **Artifacts Available** badge (96%).
- 46 artifacts received the **Artifacts Functional** badge (94%).
- 39 artifacts received the **Results Reproduced** badge (80%).

The papers that describe these artifacts can be easily recognized by the USENIX artifact evaluation badges that appear on their initial pages.

Takeaways

Cloud Resources: Increasingly, systems papers present experimental results that depend on large-scale pools of resources for reproduction. Based on feedback from the SOSP '19 efforts, we sought out resources to evaluate these types of artifacts, and Microsoft generously donated resource credits for running artifacts on Azure to help with this issue. Unfortunately, these resources were hard to leverage for the artifacts for which they would have been most useful. Several artifacts relied on access to high-end GPU resources; allocating these resources in Azure requires special approval and quota increases, which we were not able to secure. In some cases, the authors of these artifacts were able to provide reviewers with access to pre-existing resources that the AEC could use for reproduction. In the future, it may make sense to secure quotas for the use of specialized resources (specific GPUs, for example) before the start of the artifact evaluation process, based on types of resources required in the set of accepted papers.

Single vs. Multiple Badges: Of the 48 papers that received badges, 11 papers received a subset of the three available badges. We believe this is a strong outcome in favor of the multi-badge badge process we used. If we had opted to use a single badge that encompassed all of our evaluation criteria, it is likely that fewer papers would have received that badge, and consequently, fewer high-quality systems artifacts would have been recognized and documented.

Closing

We thank the authors of the 49 submitted artifacts for their hard work in creating these valuable accompaniments to their papers. We also thank the 40 AEC members, who collectively spent hundreds of hours evaluating and discussing these artifacts. Finally, we thank Microsoft for their generous support of the AEC through Microsoft Azure credits. Our hope is that the AEC effort has strengthened the work of the authors who participated, and that it will help facilitate work that builds on the papers that appear in the OSDI '20 proceedings.

The full results and badging for OSDI '20, as well as reports from other artifact evaluation processes within the systems community, can be found online at sysartifacts.github.io.

Eric Eide, *University of Utah*
Ryan Stutsman, *University of Utah*
Anjo Vahldiek-Oberwagner, *Intel Labs*
OSDI '20 Artifact Evaluation Committee Co-Chairs

OSDI '20: 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

Wednesday, November 4

Correctness

Theseus: an Experiment in Operating System Structure and State Management 1
Kevin Boos, *Rice University;* Namitha Liyanage, *Yale University;* Ramla Ijaz, *Rice University;* Lin Zhong, *Yale University*

RedLeaf: Isolation and Communication in a Safe Operating System 21
Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, and Zhaofeng Li, *University of California, Irvine;* Gerd Zellweger, *VMware Research;* Anton Burtsev, *University of California, Irvine*

Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel 41
Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang, *University of Washington*

Cobra: Making Transactional Key-Value Stores Verifiably Serializable 63
Cheng Tan and Changgeng Zhao, *NYU;* Shuai Mu, *Stony Brook University;* Michael Walfish, *NYU*

Determinizing Crash Behavior with a Verified Snapshot-Consistent Flash Translation Layer 81
Yun-Sheng Chang, Yao Hsiao, Tzu-Chi Lin, Che-Wei Tsao, Chun-Feng Wu, Yuan-Hao Chang, Hsiang-Shang Ko, and Yu-Fang Chen, *Institute of Information Science, Academia Sinica, Taiwan*

Storage Systems are Distributed Systems (So Verify Them That Way!) 99
Travis Hance, *Carnegie Mellon University;* Andrea Lattuada, *ETH Zurich;* Chris Hawblitzel, *Microsoft Research;* Jon Howell and Rob Johnson, *VMware Research;* Bryan Parno, *Carnegie Mellon University*

Storage

Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache 117
Xingda Wei, Rong Chen, and Haibo Chen, *Shanghai Jiao Tong University*

CrossFS: A Cross-layered Direct-Access File System 137
Yujie Ren, *Rutgers University;* Changwoo Min, *Virginia Tech;* Sudarsun Kannan, *Rutgers University*

From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees 155
Yifan Dai, Yien Xu, Aishwarya Ganesan, and Ramnatthan Alagappan, *University of Wisconsin - Madison;* Brian Kroth, *Microsoft Gray Systems Lab;* Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, *University of Wisconsin - Madison*

LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network 173
Mingzhe Hao, Levent Toksoz, and Nanqinqin Li, *University of Chicago;* Edward Edberg Halim, *Surya University;* Henry Hoffmann and Haryadi S. Gunawi, *University of Chicago*

A large scale analysis of hundreds of in-memory cache clusters at Twitter 191
Juncheng Yang, *Carnegie Mellon University;* Yao Yue, *Twitter;* K. V. Rashmi, *Carnegie Mellon University*

Generalized Sub-Query Fusion for Eliminating Redundant I/O from Big-Data Queries 209
Partho Sarthi, Kaushik Rajan, and Akash Lal, *Microsoft Research India;* Abhishek Modi, Prakhar Jain, Mo Liu, and Ashit Gosalia, *Microsoft;* Saurabh Kalikar, *Intel*

OS & Networking

A Simpler and Faster NIC Driver Model for Network Functions 225
Solal Pirelli and George Candea, *EPFL*

PANIC: A High-Performance Programmable NIC for Multi-tenant Networks 243
Jiaxin Lin, *University of Wisconsin - Madison;* Kiran Patel and Brent E. Stephens, *University of Illinois at Chicago;* Anirudh Sivaraman, *New York University (NYU);* Aditya Akella, *University of Wisconsin - Madison*

***Semeru*: A Memory-Disaggregated Managed Runtime** **261**
Chenxi Wang, Haoran Ma, Shi Liu, and Yuanqi Li, *UCLA;* Zhenyuan Ruan, *MIT;* Khanh Nguyen, *Texas A&M University;* Michael D. Bond, *Ohio State University;* Ravi Netravali, Miryung Kim, and Guoqing Harry Xu, *UCLA*

Caladan: Mitigating Interference at Microsecond Timescales **281**
Joshua Fried and Zhenyuan Ruan, *MIT CSAIL;* Amy Ousterhout, *UC Berkeley;* Adam Belay, *MIT CSAIL*

Overload Control for *µ*s-scale RPCs with Breakwater **299**
Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay, *MIT CSAIL*

AIFM: High-Performance, Application-Integrated Far Memory **315**
Zhenyuan Ruan, *MIT CSAIL;* Malte Schwarzkopf, *Brown University;* Marcos K. Aguilera, *VMware Research;* Adam Belay, *MIT CSAIL*

Consistency

Performance-Optimal Read-Only Transactions **333**
Haonan Lu, *Princeton University;* Siddhartha Sen, *Microsoft Research;* Wyatt Lloyd, *Princeton University*

Toward a Generic Fault Tolerance Technique for Partial Network Partitioning **351**
Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany, *University of Waterloo, Canada*

***PACEMAKER*: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy** **369**
Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger, *Carnegie Mellon University*

Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories **387**
Jialin Li, *National University of Singapore;* Jacob Nelson, *Microsoft Research;* Ellis Michael, *University of Washington;* Xin Jin, *Johns Hopkins University;* Dan R. K. Ports, *Microsoft Research*

FlightTracker: Consistency across Read-Optimized Online Stores at Facebook **407**
Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson, *Facebook, Inc.*

KVell+: Snapshot Isolation without Snapshots **425**
Baptiste Lepers and Oana Balmau, *University of Sydney;* Karan Gupta, *Nutanix Inc.;* Willy Zwaenepoel, *University of Sydney*

Thursday, November 5

Machine Learning 1

Serving DNNs like Clockwork: Performance Predictability from the Bottom Up **443**
Arpan Gujarati, *Max Planck Institute for Software Systems;* Reza Karimi, *Emory University;* Safya Alzayat, Wei Hao, and Antoine Kaufmann, *Max Planck Institute for Software Systems;* Ymir Vigfusson, *Emory University;* Jonathan Mace, *Max Planck Institute for Software Systems*

A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters **463**
Yimin Jiang, *Tsinghua University and ByteDance;* Yibo Zhu, *ByteDance;* Chang Lan, *Google;* Bairen Yi, *ByteDance;* Yong Cui, *Tsinghua University;* Chuanxiong Guo, *ByteDance*

Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads **481**
Deepak Narayanan and Keshav Santhanam, *Stanford University and Microsoft Research*; Fiodar Kazhamiaka, *Stanford University;* Amar Phanishayee, *Microsoft Research;* Matei Zaharia, *Stanford University*

PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications **499**
Zhihao Bai and Zhen Zhang, *Johns Hopkins University;* Yibo Zhu, *ByteDance Inc.;* Xin Jin, *Johns Hopkins University*

HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees **515**
Hanyu Zhao, *Peking University and Microsoft;* Zhenhua Han, *The University of Hong Kong and Microsoft;* Zhi Yang, *Peking University;* Quanlu Zhang, Fan Yang, Lidong Zhou, and Mao Yang, *Microsoft;* Francis C.M. Lau, *The University of Hong Kong;* Yuqi Wang, Yifan Xiong, and Bin Wang, *Microsoft*

AntMan: Dynamic Scaling on GPU Clusters for Deep Learning **533**
Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia, *Alibaba Group*

Consensus

Write Dependency Disentanglement with HORAE **549**
Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu, *Tsinghua University*

Blockene: A High-throughput Blockchain Over Mobile Devices **567**
Sambhav Satija and Apurv Mehra, *Microsoft Research India;* Sudheesh Singanamalla, *University of Washington;* Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam, *Microsoft Research India*

Tolerating Slowdowns in Replicated State Machines using Copilots **583**
Khiem Ngo, *Princeton University;* Siddhartha Sen, *Microsoft Research;* Wyatt Lloyd, *Princeton University*

Microsecond Consensus for Microsecond Applications **599**
Marcos K. Aguilera and Naama Ben-David, *VMware Research;* Rachid Guerraoui, *EPFL;* Virendra J. Marathe, *Oracle Labs;* Athanasios Xygkis and Igor Zablotchi, *EPFL*

Virtual Consensus in Delos **617**
Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song, *Facebook, Inc*.

Byzantine Ordered Consensus without Byzantine Oligarchy **633**
Yunhao Zhang, *Cornell University;* Srinath Setty, Qi Chen, and Lidong Zhou, *Microsoft Research;* Lorenzo Alvisi, *Cornell University*

Bugs

From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes **651**
Florian Rommel and Christian Dietrich, *Leibniz Universität Hannover;* Daniel Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, and Olaf Spinczyk, *Universität Osnabrück;* Daniel Lohmann, *Leibniz Universität Hannover*

Testing Database Engines via Pivoted Query Synthesis **667**
Manuel Rigger and Zhendong Su, *ETH Zurich*

Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing **683**
Fabian Ruffy, Tao Wang, and Anirudh Sivaraman, *New York University*

***Aragog:* Scalable Runtime Verification of Shardable Networked Systems** **701**
Nofel Yaseen, *University of Pennsylvania;* Behnaz Arzani and Ryan Beckett, *Microsoft Research;* Selim Ciraci, *Microsoft;* Vincent Liu, *University of Pennsylvania*

Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution **719**
Yigong Hu, Gongqi Huang, and Peng Huang, *Johns Hopkins University*

Testing Configuration Changes in Context to Prevent Production Failures **735**
Xudong Sun, Runxiang Cheng, Jianyan Chen, and Elaine Ang, *University of Illinois at Urbana-Champaign;* Owolabi Legunsen, *Cornell University;* Tianyin Xu, *University of Illinois at Urbana-Champaign*

Scheduling

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms **753**
Pradeep Ambati, *University of Massachusetts, Amherst;* Íñigo Goiri, Felipe Frujeri, *Microsoft Azure and Microsoft Research;* Alper Gun and Ke Wang, *Google;* Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini, *Microsoft Azure and Microsoft Research*

The *CacheLib* Caching Engine: Design and Experiences at Scale **769**
Benjamin Berg, *Carnegie Mellon University;* Daniel S. Berger, *Carnegie Mellon University and Microsoft Research;* Sara McAllister and Isaac Grosof, *Carnegie Mellon University;* Sathya Gunasekar, Jimmy Lu, Michael Uhlar, and Jim Carrig, *Facebook;* Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger, *Carnegie Mellon University*

Twine: A Unified Cluster Management System for Shared Infrastructure **787**
Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang, *Facebook Inc*.

FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices 805
Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer, *University of Illinois at Urbana–Champaign*

Building Scalable and Flexible Cluster Managers Using Declarative Programming 827
Lalith Suresh, *VMware;* João Loff, *IST* (*ULisboa*) / *INESC-ID;* Faria Kalim, *UIUC;* Sangeetha Abdu Jyothi, *UC Irvine and VMware;* Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch, *VMware*

Protean: VM Allocation Service at Scale 845
Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda, *Microsoft Azure and Microsoft Research*

Friday, November 6

Machine Learning 2

Ansor: Generating High-Performance Tensor Programs for Deep Learning 863
Lianmin Zheng, *UC Berkeley;* Chengfan Jia, Minmin Sun, and Zhao Wu, *Alibaba Group;* Cody Hao Yu, *Amazon Web Services, Inc;* Ameer Haj-Ali, *UC Berkeley;* Yida Wang, *Amazon Web Services;* Jun Yang, *Alibaba Group;* Danyang Zhuo, *UC Berkeley and Duke University;* Koushik Sen, Joseph E. Gonzalez, and Ion Stoica, *UC Berkeley*

RAMMER: Enabling Holistic Deep Learning Compiler Optimizations with *r*Tasks 881
Lingxiao Ma, *Peking University and Microsoft Research;* Zhiqiang Xie, *ShanghaiTech University and Microsoft Research;* Zhi Yang, *Peking University;* Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou, *Microsoft Research*

A Tensor Compiler for Unified Machine Learning Prediction Serving 899
Supun Nakandala, *UC San Diego;* Karla Saur, *Microsoft;* Gyeong-In Yu, *Seoul National University;* Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi, *Microsoft*

Retiarii: A Deep Learning Exploratory-Training Framework 919
Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou, *Microsoft Research*

KungFu: Making Training in Distributed Machine Learning Adaptive 937
Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch, *Imperial College London*

Hardware

FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization 955
Dongup Kwon, *Department of Electrical and Computer Engineering, Seoul National University / Memory Solutions Lab, Samsung Semiconductor Inc.;* Junehyuk Boo and Dongryeong Kim, *Department of Electrical and Computer Engineering, Seoul National University;* Jangwoo Kim, *Department of Electrical and Computer Engineering, Seoul National University / Memory Solutions Lab, Samsung Semiconductor Inc*.

hXDP: Efficient Software Packet Processing on FPGA NICs 973
Marco Spaziani Brunella and Giacomo Belocchi, *Axbryd/University of Rome Tor Vergata;* Marco Bonola, *Axbryd/CNIT;* Salvatore Pontarelli, *Axbryd;* Giuseppe Siracusano, *NEC Laboratories Europe;* Giuseppe Bianchi, *University of Rome Tor Vergata;* Aniello Cammarano, Alessandro Palumbo, and Luca Petrucci, *CNIT/University of Rome Tor Vergata;* Roberto Bifulco, *NEC Laboratories Europe*

Do OS abstractions make sense on FPGAs? 991
Dario Korolija, Timothy Roscoe, and Gustavo Alonso, *ETH Zurich*

Assise: Performance and Availability via Client-local NVM in a Distributed File System 1011
Thomas E. Anderson, *University of Washington;* Marco Canini, *KAUST;* Jongyul Kim, *KAIST;* Dejan Kostić, *KTH Royal Institute of Technology;* Youngjin Kwon, *KAIST;* Simon Peter, *The University of Texas at Austin;* Waleed Reda, *KTH Royal Institute of Technology and Université catholique de Louvain;* Henry N. Schuh, *University of Washington;* Emmett Witchel, *The University of Texas at Austin*

Persistent State Machines for Recoverable In-memory Storage Systems with NVRam 1029
Wen Zhang, *UC Berkeley;* Scott Shenker, *UC Berkeley/ICSI;* Irene Zhang, *Microsoft Research/University of Washington*

AGAMOTTO: How Persistent is your Persistent Memory Application? 1047
Ian Neal, Ben Reeves, Ben Stoler, and Andi Quinn, *University of Michigan;* Youngjin Kwon, *KAIST;* Simon Peter, *University of Texas at Austin;* Baris Kasikci, *University of Michigan*

Security

Orchard: Differentially Private Analytics at Scale 1065
Edo Roth, Hengchu Zhang, Andreas Haeberlen, and Benjamin C. Pierce, *University of Pennsylvania*

Achieving 100Gbps Intrusion Prevention on a Single Server 1083
Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry, *Carnegie Mellon University*

DORY: An Encrypted Search System with Distributed Trust 1101
Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica, *University of California, Berkeley*

SafetyPin: Encrypted Backups with Human-Memorable Secrets 1121
Emma Dauterman, *UC Berkeley;* Henry Corrigan-Gibbs, *EPFL and MIT CSAIL;* David Mazières, *Stanford University*

Efficiently Mitigating Transient Execution Attacks using the Unmapped Speculation Contract 1139
Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich, *MIT CSAIL*

Clusters

Predictive and Adaptive Failure Mitigation to Avert Production Cloud VM Interruptions 1155
Sebastien Levy, Randolph Yao, Youjiang Wu, and Yingnong Dang, *Microsoft Azure;* Peng Huang, *Johns Hopkins University;* Zheng Mu, *Microsoft Azure;* Pu Zhao, *Microsoft Research;* Tarun Ramani, Naga Govindaraju, and Xukun Li, *Microsoft Azure;* Qingwei Lin, *Microsoft Research;* Gil Lapid Shafriri and Murali Chintalapati, *Microsoft Azure*

Sundial: Fault-tolerant Clock Synchronization for Datacenters 1171
Yuliang Li, *Google Inc. and Harvard University;* Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, and Dave Platt, *Google Inc.;* Simon Sabato, *Lilac Cloud;* Minlan Yu, *Harvard University;* Nandita Dukkipati, Prashant Chandra, and Amin Vahdat, *Google Inc.*

Fault-tolerant and transactional stateful serverless workflows 1187
Haoran Zhang, *University of Pennsylvania;* Adney Cardoza, *Rutgers University–Camden;* Peter Baile Chen, Sebastian Angel, and Vincent Liu, *University of Pennsylvania*

Unearthing inter-job dependencies for better cluster scheduling 1205
Andrew Chung, *Carnegie Mellon University;* Subru Krishnan, Konstantinos Karanasos, and Carlo Curino, *Microsoft;* Gregory R. Ganger, *Carnegie Mellon University*

RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers 1225
Hang Zhu, *Johns Hopkins University;* Kostis Kaffes, *Stanford University;* Zixu Chen, *Johns Hopkins University;* Zhenming Liu, *College of William and Mary;* Christos Kozyrakis, *Stanford University;* Ion Stoica, *UC Berkeley;* Xin Jin, *Johns Hopkins University*

Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale 1241
Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan, *Google LLC*



Theseus: an Experiment in Operating System Structure and State Management

Kevin Boos
Rice University

Namitha Liyanage
Yale University

Ramla Ijaz
Rice University

Lin Zhong
Yale University

Abstract

This paper describes an operating system (OS) called Theseus. Theseus is the result of multi-year experimentation to redesign and improve OS modularity by reducing the states one component holds for another, and to leverage a safe programming language, namely Rust, to shift as many OS responsibilities as possible to the compiler.

Theseus embodies two primary contributions. First, an OS structure in which many tiny components with clearly-defined, runtime-persistent bounds interact without holding states for each other. Second, an intralingual approach that realizes the OS itself using language-level mechanisms such that the compiler can enforce invariants about OS semantics.

Theseus's structure, intralingual design, and state management realize live evolution and fault recovery for core OS components in ways beyond that of existing works.

1 Introduction

We report an experimentation of OS structural design, state management, and implementation techniques that leverage the power of modern safe systems programming languages, namely Rust. This endeavor was initially motivated by studies of *state spill* [16]: one software component harboring changed states as a result of handling an interaction from another component, such that their future correctness depends on said states. Prevalent in modern systems software, state spill leads to fate sharing between otherwise modularized and isolated components and thus hinders the realization of desirable computing goals such as evolvability and availability. For example, state spill in Android system services causes the entire userspace frameworks to crash upon a system service failure, losing the states and progress of all applications, even those not using the failed service [16]. Reliable microkernels further attest that management of states spilled into OS services is a barrier to fault tolerance [21] and live update [28].

Evolvability and availability of systems software are crucial in environments where reliability is necessary yet hardware redundancy is expensive or impossible. For example, systems software updates must be painstakingly applied without downtime or lost execution context in pacemakers [26] and space probes [25, 62]. Even in datacenters, where network switches are replicated for reliability, switch software failures and maintenance updates still lead to network outages [27,48].

On the quest to determine to what extent state spill can be avoided in OS code, we chose to write an OS from scratch. We were drawn to Rust because its ownership model provides a convenient mechanism for implementing isolation and zero-cost state transfer between OS components. Our initial OS-building experience led to two important realizations. First, mitigating state spill, or better state management in general, necessitates a rethinking of OS structure because state spill (by definition) depends on how the OS is modularized. Second, modern systems programming languages like Rust can be used not just to write safe OS code but also to statically ensure certain correctness invariants for OS behaviors.

The outcome of our experimentation is Theseus OS, which makes two contributions to systems software design and implementation. First, Theseus has a novel OS structure of many tiny components with clearly-defined, runtime-persistent bounds. The system maintains metadata about and tracks interdependencies between components, which facilitates live evolution and fault recovery of these components (§3).

Second, and more importantly, Theseus contributes the *intralingual* OS design approach, which entails matching the OS's execution environment to the runtime model of its implementation language and implementing the OS itself using language-level mechanisms. Through intralingual design, Theseus empowers the compiler to apply its safety checks to OS code with no gaps in its understanding of code behavior, and shifts semantic errors from runtime failures into compile-time errors, both to a greater degree than existing OSes. Intralingual design goes beyond safety, enabling the compiler to statically check OS semantic invariants and assume resource bookkeeping duties. This is elaborated in §4.

Theseus's structure and intralingual design naturally reduce states the OS must maintain, reducing state spill between its components. We describe Theseus's state management techniques to further mitigate the effects of state spill in §5.

To demonstrate the utility of Theseus's design, we implement live evolution and fault recovery (for availability) within it (§6). With this, we posit that Theseus is well-suited for high-end embedded systems and datacenter components, where availability is needed in the absence of or in addition to hardware redundancy. Therein, Theseus's limitations of being a new OS and needing safe-language programs have a lesser impact, as applications can be co-developed with the OS in an environment under a single operator's control.

We evaluate how well Theseus achieves these goals in §7. Through a set of case studies, we show that Theseus can easily and arbitrarily live evolve core system components in ways beyond prior live update works, e.g., joint application-kernel evolution, or evolution of microkernel-level components. As Theseus can gracefully handle language-level faults (`panic`s in Rust), we demonstrate Theseus's ability to tolerate more challenging transient hardware faults that manifest in the OS core. To this end, we present a study of fault manifestation and recovery in Theseus and a comparison with MINIX 3 of fault recovery for components that necessarily exist inside the microkernel. Although performance is not a primary goal of Theseus, we find that its intralingual and spill-free designs do not impose a glaring performance penalty, but that the impact varies across subsystems.

Theseus is currently implemented on x86_64 with support for most hardware features, such as multicore processing, preemptive multitasking, SIMD extensions, basic networking and disk I/O, and graphical displays. It represents roughly four person-years of effort and comprises ~38000 lines of from-scratch Rust code, 900 lines of bootstrap assembly code, 246 crates of which 176 are first-party, and 72 unsafe code blocks or statements across 21 crates, most of which are for port I/O or special register access.

However, Theseus is far less complete than commercial systems, or experimental ones such as Singularity [33] and Barrelfish [8] that have undergone substantially more development. For example, Theseus currently lacks POSIX support and a full standard library. Thus, we do not make claims about certain OS aspects, e.g., efficiency or security; this paper focuses on Theseus's structure and intralingual design and the ensuing benefits for live evolution and fault recovery.

Theseus's code and documentation are open-source [61].

2 Rust Language Background

The Rust programming language [40] is designed to provide strong type and memory safety guarantees at compile time, combining the power and expressiveness of a high-level managed language with the C-like efficiency of no garbage collection or underlying runtime. Theseus leverages many Rust features to realize an intralingual, safe OS design and employs the *crate*, Rust's project container and translation unit, for source-level modularity. A crate contains source code and a dependency manifest. Theseus does not use Rust's standard library but does use its fundamental `core` and `alloc` libraries.

Rust's *ownership* model is the key to its compile-time memory safety and management. Ownership is based on affine types, in which a value can be used at most once. In Rust, every value has an owner, e.g., the string value `"hello!"` allocated in L4 below is owned by the `hello` variable. After a value is moved, e.g., if `"hello!"` was moved in L5 from `hello` to `owned_string` (L14), its ownership would be transferred and the previous owner (`hello`) could no longer use it.

```
1  fn main() {
2     let hel: &str;
3     {
4        let hello = String::from("hello!");
5        // consume(hello); // ⟶ "value moved" error in L6
6        let borrowed_str: &str = &hello;
7        hel = substr(borrowed_str);
8     }
9     // print!("{}", hel); // ⟶ lifetime error
10 }
11 fn substr<'a>(input_str: &'a str) -> &'a str {
12    &input_str[0..3]  // return value has lifetime 'a
13 }
14 fn consume(owned_string: String) {...}
```

When the owner's scope ends, e.g., at the end of a lexical block, the owned value is *dropped* (released) by virtue of the compiler inserting a call to its destructor. Destructors in Rust are realized by implementing the `Drop` trait for a given type, in which a custom *drop handler* can perform arbitrary actions beyond freeing memory. On L8 above, the `hello` string falls out of scope and is auto-deallocated by its drop handler.

Values can also be *borrowed* to obtain references to them (L6), and the lifetime of those references cannot outlast the lifetime of the owned value. The syntax in L11 gives the name `'a` to the lifetime of the `input_str` argument, and specifies that the returned `&str` reference has that same lifetime `'a`. That returned `&str` reference is assigned to `hel` in L7, which would result in a lifetime violation in L9 because `hel` would be used *after* the owned value it was originally borrowed from (`hello`) was dropped in L8. Rust's compiler includes a borrow checker to enforce these lifetime rules, as well as the core tenet of *aliasing XOR mutability*, in which there can be multiple immutable references or a single mutable reference to a value, but not both at once. This allows it to statically ensure memory safety for values on the stack and heap.

Theseus also extensively leverages Rust *traits*, a declaration of an abstract type that specifies the set of methods the type must implement, similar to polymorphic interfaces in OOP languages. Traits can be used to place *bounds* on generic type parameters. For example, the function `fn print_str<T: Into<String>>(s: T){ }` uses the underlined trait bound to specify that its argument named `s` must be of any abstract type `T` that can be converted into a `String`.

3 Theseus Overview and Design Principles

The overall design of Theseus specifies a system architecture consisting of many small distinct components, called *cells*, which can be composed and interchanged at runtime. A cell is a software-defined unit of modularity that serves as the core building block of the OS, much like their namesake of biological cells in an organism (no relation to Rust's `std::cell`). Theseus enables all software written in safe Rust, including applications and libraries, to coexist alongside the core OS components in a single address space (SAS) and execute at a single privilege level (SPL), building upon language-provided type and memory safety to realize isolation instead of hardware protection. Everything presented herein is written in Rust and runs in the SAS/SPL environment.

Theseus follows three design principles:

P1. Require *runtime-persistent* bounds for *all* cells.

P2. Maximize the power of the language and compiler.

P3. Minimize *state spill* between cells.

The remainder of this section describes how Theseus satisfies the first principle and why it matters, while §4 and §5 discuss the second and third principles, respectively.

3.1 Structure of Runtime-Persistent Cells

Cells in Theseus have bounds that are clearly defined at implementation time and persist into and throughout runtime: a cell exists as a Rust *crate* at implementation time, a single object file at compile time, and a set of loaded memory regions with per-section bounds and dependency metadata at runtime. This applies to *all* cells, not just a select subset such as kernel extensions in monolithic and safe-language OSes or userspace servers in microkernels; there are no exemptions for components within a "base kernel" image. Explicit cell bounds identifiable at runtime are the foundation for strong data/fault isolation and state management in Theseus.

At runtime, Theseus loads and links *all* cells into the system on demand. Briefly, this entails finding and parsing the cell object file, loading its sections into memory, resolving its dependencies to write linker relocation entries, recursively loading any missing cells as needed, and adding new public symbols to a symbol map. In doing so, Theseus constructs detailed *cell metadata*, depicted in Figure 1, which is crucial knowledge for live evolution (§6.1) and fault recovery (§6.2). The set of loaded cells defines a CellNamespace, a true namespace containing all cells' public symbols, used to quickly resolve dependencies between cells. Each loaded cell node tracks its constituent sections and the memory regions (§4.3.1) that contain them. The sections in each cell correspond to those in its crate's object file, e.g., executable, read-only data, and read-write data sections. Each loaded section node tracks its size, location in memory, and bidirectional dependencies (incoming and outgoing); additional metadata exists to accelerate cell swapping and other system functions.

Persistence of Cell Bounds Reduces Complexity: Theseus's persistent cell bounds provide a consistent abstraction of OS structure throughout all phases of their existence. This reduces the complexity of a developer's mental model of the OS and simplifies fault recovery and evolution logic, as Theseus can introspect upon and manage its own code from the same cell-oriented viewpoint at runtime. The SAS/SPL environment augments this consistent view with *completeness*, in that everything from top-level applications and libraries to core kernel components are observable as cells. This enables Theseus to (*i*) implement a single mechanism, cell swapping, uniformly applicable to *any* cell, and (*ii*) jointly evolve cells from multiple system layers (e.g., applications and kernel components) in a safe manner.

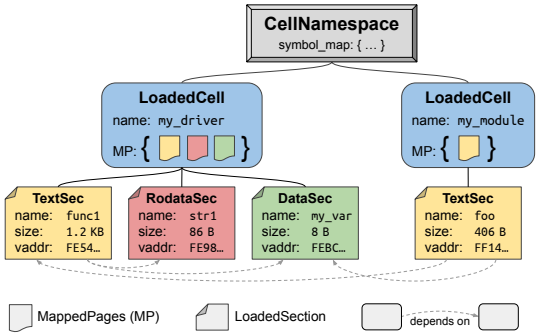


Figure 1: Theseus constructs detailed metadata that tracks runtime cell bounds in memory and bidirectional, per-section dependencies in order to simplify cell swapping logic.

Striking a Balance with Cell Granularity: Theseus cells are elementary in their scope; we follow separation of concerns to split functionality into many tiny crates, letting unavoidable circular dependencies between them halt further decomposition. We do not use Rust's source-level module hierarchy in which one crate contains multiple Rust (sub)modules, as those module bounds are lost when the crate is built into an object file. Instead, we extract would-be modules into distinct crates, realizing hierarchy by organizing crates' source files into folders in Theseus's repository. This design offers both a programmer-friendly hierarchical view of source code and a simple system view of all cells as a flat set of distinct object files. It also strikes a balance between the complexity of needing to swap myriad tiny cells and the inefficiency and impracticality of swapping a large monolithic cell.

3.2 Bootstrapping Theseus with the `nano_core`

Theseus splits the compilation process at the linker stage, placing raw cell object files directly into the OS image such that linkage is deferred to runtime. From a practical standpoint, unlinked object files cannot run, so we must jump-start Theseus with the `nano_core`. The `nano_core` is a set of normal cells statically linked together into a tiny, executable "base kernel" image, comprising only components needed to bootstrap a bare-minimum environment that supports virtual memory and loading/linking object files. Because statically linking cells loses their bounds and dependencies, the `nano_core` fully replaces itself at the final bootstrap stage by dynamically loading its constituent cells one by one, using augmented symbol tables and other metadata burned into the OS image at build time. This meets the requirement of runtime-persistent bounds for *all* cells, allowing the `nano_core` to be safely unloaded after bootstrap.

4 Power to the Language

The second design principle Theseus follows is to leverage the power of the language by enabling the compiler to check safety and correctness invariants to the fullest extent possible. We term this approach *intralingual*, within the language, as it

involves matching Theseus's execution environment to that of the language's runtime model, and implementing OS semantics fully within the strong, static type system offered by modern languages like Rust. This extends compiler-checked invariants (e.g., no dangling references) to *all* types of resources, not just those built into the language.

Intralingual design offers two primary benefits. First, it empowers the compiler to take over resource management duties, reducing the states the OS must maintain, which in turn reduces state spill and strengthens isolation. Second, it enables the compiler to apply safety checks with no gaps in its understanding of code behavior, approaching end-to-end safety from applications to core kernel components and shifting semantic runtime errors into compile-time errors.

In contrast, traditional *extralingual* approaches rely on hardware protection and runtime checks to uphold invariants for safety, isolation, and correctness. These features are transparent to the compiler and require unsafe code. Even existing safe-language OSes [3, 13, 33, 44] have a gap between language-level safe code and the underlying unsafe core that implements the language's required abstractions as a black box. Below, we describe how Theseus closes this gap and opens up such black boxes to the compiler.

4.1 Matching the Language's Runtime Model

The compiler for many languages, including Rust, expects that its output will become (part of) an executable that runs within one address space and privilege level, e.g., a single userspace process. Thus, the compiler cannot holistically observe or check the behavior of independently-compiled components that run in different address spaces or privilege levels.

To address this shortcoming, we tailor Theseus's OS execution environment to match Rust's runtime model: (*i*) only a single address space (SAS) exists and thus a single set of addresses is visible, for which Theseus guarantees a one-to-one virtual-to-physical mapping; (*ii*) all code executes within a single privilege level (SPL), thus there is no other world or mode of execution; (*iii*) only a single allocator instance exists, matching the compiler's expectation that a global heap serves all allocation requests. Note that Theseus does support multiple arbitrary heaps within that single instance (§7.3).

4.2 Intralingual OS Design

Matching the language's runtime model only allows the compiler to *view* all Theseus components. For the compiler to *understand* those components and apply its safety checks to them, we must implement them in a manner that exposes their safety requirements, invariants, and semantics to the compiler. As an aside, Theseus uses safe code to the fullest extent possible at all layers of the system, prioritizing safety over all else, e.g., convenience, performance. It only descends into unsafety when fundamentally unavoidable: executing instructions *directly* above hardware and select functions within Rust's foundational libraries, i.e., `core` and `alloc`.

Theseus goes beyond language safety to further empower the compiler to check our custom OS invariants as if they were built in. First, for each OS resource, Theseus identifies the set of invariants that prevent unsafety and incorrect usage. As the Rust compiler already checks myriad invariants for the usage of language-provided types and mechanisms, Theseus employs these existing mechanisms to allow its resource-specific invariants to be *subsumed* into those compiler invariants. For example, Theseus uses Rust's built-in reference types, such as `&T` and `Arc<T>` (Atomic Reference-Counted pointer), to share resources (e.g., memory regions, channel endpoints) across multiple tasks in a safe language-level manner, instead of extralingual sharing mechanisms like raw pointers or mapping multiple pages to the same frame. This eliminates possible use-after-free errors by subsuming resource mismanagement checks into the compiler's lifetime invariants.

Second, Theseus employs *lossless* interfaces for both external functions that export a resource's semantics and internal functions that implement those semantics. An interface is lossless if crossing it preserves all language-level context, e.g., an object's type, lifetime, or ownership/borrowed status. Furthermore, the *provenance* of that language-level context must be statically determinable, such that the compiler can authenticate that there was no broken link in the chain of calls and interface crossings when using a given resource. In other words, language-level knowledge must not be lost and then reconstituted extralingually. For example, invoking a system call in Linux loses the type and lifetime information of its arguments because they must be reduced to raw integer values to cross the user-kernel boundary.

Ensuring Resource Cleanup via Unwinding

One major invariant we enforce beyond default Rust safety is to prevent *resource leakage*, an acquired resource not being released even after no references to it remain. Although leakage does not violate safety, it is generally incorrect behavior. Theseus prevents resource leakage by (*i*) implementing all cleanup semantics in drop handlers (§2), a lossless language-level approach that allows the compiler to solely determine when it is safe to trigger resource cleanup, and (*ii*) employing *stack unwinding* to ensure acquired resources are always released in both normal and exceptional execution.

When tasks acquire resources in Theseus, they directly own objects representing those resources on their stack (§5.1). The Rust compiler tracks ownership of those objects to statically determine when a resource is dropped and, thus, where to insert its cleanup routine. Implementing all resource cleanup in only drop handlers frees developers from the burden of correctly ordering release operations or considering corner cases such as exceptional control flow jumps. Applying this to acquired locks allows Theseus to statically prevent many cases of deadlock: lock guards are auto-released during unwinding, and domain-specific locks automatically disable/re-enable preemption or interrupts, e.g., when modifying task runstates.

We implement Theseus's *unwinder* from scratch in Rust, with custom unwinding logic based on the DWARF standard [1] but independent from existing unwind libraries; thus, it works in core OS contexts without a standard library or allocation. Theseus starts the unwinder only upon a software or hardware exception or a request to kill a task; it does not interfere with normal execution performance, unlike garbage collectors. This prevents failed or uncooperative tasks from jeopardizing resource release and reclamation, strengthening fault isolation. The unwinder uses compiler-emitted information along with cell metadata to locate previous frames in the call stack, calculate and restore register values present during that frame, and discover and invoke cleanup routines or exception-catching blocks. Cell metadata even enables the unwinder to traverse through nonstandard stack frames for hardware-entered asynchronous calling contexts, e.g., interrupts or CPU exception handlers.

Theseus supports intralingual resource revocation in two forms. First, Theseus can forcibly revoke generic resources by killing and unwinding an uncooperative task. This avoids isolation-breaking undefined behavior by ceasing to execute a task once its assumptions of safe resource access no longer hold. Second, Theseus can cooperatively revoke reclaimable resources, such as in-memory caches and buffer pools, which express the possibility of resource absence within their type definition, e.g., using `Option` or weak references. This design unifies system-level and language-level resource actions to guarantee that revoked resources are freed exactly once.

4.3 Examples of Intralingual Subsystems

We next describe how Theseus intralingually implements foundational OS resources, namely memory management and task management. Additional invariants, details, and examples, such as inter-task communication (ITC) channels, are omitted for brevity and available elsewhere [15].

4.3.1 Memory Management

Theseus intralingually implements virtual memory via the `MappedPages` type of Listing 2, which represents a region of virtually-contiguous pages statically guaranteed to be mapped to (optionally contiguous) real physical frames. `MappedPages` is the fundamental, sole way to map and access memory in Theseus, and serves as the backing representation for stacks, heaps, and arbitrary memory regions, e.g., device MMIO and loaded cells. The design of `MappedPages` empowers the compiler's type system to enforce the following key invariants, extending Rust's memory safety checks to *all* OS memory regions, not just the compiler-known stack and heap.

M.1: *The mapping from virtual pages to physical frames must be one-to-one, or bijective.* This prevents aliasing (sharing) from occurring beneath the language, forcing all shared memory access in Theseus to use *only* language-level mechanisms, such as references (`&MappedPages`). In Theseus's SAS environment (§4.1), this is both possible and non-restrictive. In contrast, both conventional and existing safe-language OS

```
1  fn main() -> Result<()> {
2    let frames = get_hpet_frames()?;
3    let pages = allocate_pages(frames.count())?;
4    let mp_pgs = map(pages, frames, flags, pg_tbl)?;
5    {
6      let hpet: &HpetRegisters = mp_pgs.as_type(0)?;
7      print!("HPET device Vendor ID: {}", hpet.caps_id.read() >> 16);
8    }
9    let (sender, receiver) = rendezvous::new_channel::<MappedPages>();
10   let new_task = spawn_task(receiver_task, receiver)?;
11   sender.send(mp_pgs)?;
12   Ok(()) // `mp_pgs` not dropped, it was moved
13 }
14 fn receiver_task(receiver: Receiver<MappedPages>) -> Result<()> {
15   let mp: MappedPages = receiver.receive()?;
16   let hpet: &HpetRegisters = mp.as_type(0)?;
17   print!("Current HPET ticks: {}", hpet.main_counter.read());
18   Ok(())  // `mp` auto-dropped and unmapped here
19 }
20 struct HpetRegisters {
21   pub caps_and_id:     ReadOnly<u64>,
22   _padding:            [u64, ...],
23   pub main_counter:    Volatile<u64>,
24   ...
25 }
```

Listing 1: Example code that maps a memory region representing the HPET device, accesses the HPET vendor ID via MMIO, then spawns a new task and sends that memory region to it over a channel. The new task receives that memory region and uses it to read the HPET counter. This refers to code continued in Listing 2 and 3.

designs allow different virtual pages to map the same physical frame, an extralingual approach that renders sharing transparent to the compiler and thus uncheckable for safety.

We realize this invariant via the `map()` function (L26), which leverages type safety to take ownership of the allocated pages and frames in order to return a new `MappedPages` object. The lossless `map()` interface statically ensures the provenance of this relationship between `AllocatedPages`, `AllocatedFrames`, and `MappedPages`, guaranteeing they cannot be reused for duplicate mappings.

M.2: *Memory must not be accessible beyond the mapped region's bounds.* To access a memory region, one must use `MappedPages` methods like `as_type()` (L45) or `as_slice()` (L52) that overlay a statically-sized struct or dynamically-sized slice atop it; mutable versions exist, see M.4 below. The in-bounds invariant (L46) is checked dynamically unless elided when the size and offset are statically known, as in some MMIO cases. These access functions are lossless because they return sized types that preserve the lifetime relationship described below.

M.3: *A memory region must be unmapped exactly once, only after there remain no outstanding references to it.* `MappedPages` realizes its release and cleanup semantics only within its drop handler (L38), ensuring that a `MappedPages` object, such as `mp` in L15 of Listing 1, is unmapped in both normal execution (L18) and exceptional execution. Correspondingly, memory must not be accessible after it has been unmapped. The above access methods tie the lifetime of the re-typed borrowed reference `&'m T` to the lifetime of its backing `MappedPages` memory region, allowing compiler lifetime checks to statically prevent use-after-free. As such, obtaining ownership of an overlaid struct is impossible by design, as

```
26 pub fn map(pages: AllocatedPages, frames: AllocatedFrames,
              flags: EntryFlags, ...) -> Result<MappedPages> {
27   for (page, frame) in pages.iter().zip(frames.iter()) {
28     let mut pg_tbl_entry = pg_tbl.walk_to(page, flags)?
          .get_pte_mut(page.pte_offset());
29     pg_tbl_entry.set(frame.start_addr(), flags)?;
30   }
31   Ok(MappedPages { pages, frames, flags })
32 }
33 pub struct MappedPages {
34   pages:  AllocatedPages,
35   frames: AllocatedFrames,
36   flags:  EntryFlags,
37 }
38 impl Drop for MappedPages {
39   fn drop(&mut self) {
40     // unmap here: clear page table entry, invalidate TLB.
41     // AllocatedPages/Frames are auto-dropped and deallocated.
42   }
43 }
44 impl MappedPages {
45   pub fn as_type<'m, T>(&'m self, offset: usize) -> Result<&'m T> {
46     if offset + size_of::<T>() > self.size_in_bytes() {
47       return Error::OutOfBounds;
48     }
49     let typed_mem: &T = unsafe {
         &*((self.pages.start_addr() + offset) as *const T) };
50     Ok(typed_mem)
51   }
52   pub fn as_slice<'m, T>(&'m self, offset: usize, count: usize)
       -> Result<&'m [T]> { ... }
53 }
```

Listing 2: The basic `MappedPages` type (L33) exposes an interface (L44-53) for safely accessing its underlying memory region. The `map()` function (L26) maps a range of virtual pages to physical frames and returns a new `MappedPages` instance that represents that memory region. Sanity checks and details omitted for brevity.

that would lossily discard the above lifetime relationship.

M.4: *A memory region must only be mutable or executable if mapped as such.* We ensure this using dedicated types, `MappedPagesMut` and `MappedPagesExec`, that offer `as_type_mut()` and `as_function()`, which statically prevents page protection violations as described elsewhere [15].

In summary, `MappedPages` bridges the semantic gap between the compiler's and OS's knowledge of memory, guaranteeing at compile time that unexpected invalid page faults cannot occur. Note that the necessary unsafe code in L49 is *innocuous* (see §8) as it merely indicates that the compiler cannot ensure the overlaid struct type has valid contents. Correctness of struct contents (e.g., `HpetRegisters` in L20) is unavoidably left to the developer. Regardless of developer mistakes, the compiler can still check that this unsafe code does not violate fault or data isolation because other invariants ensure it cannot produce dangling references (M.3) or access out-of-bounds addresses (M.2) beyond the reach of safe code. All other memory management code is safe down to the lowest level, where page table walks require extralingual code to accommodate hardware-defined page table formats.

4.3.2 Task Management

While `MappedPages` is the center of intralingual memory management, the `Task` struct in Theseus is minimized in both content and significance. Rather, task management centers around intralingual functions that leverage a consistent set of generic type parameters to handle each stage of the task life-

```
54 pub trait TFunc<A,R> = FnOnce(A) -> R;
55 pub trait TArg = Send + 'static;
56 pub trait TRet = Send + 'static;
57 pub fn spawn_task<F,A,R>(func: F, arg: A, ...) -> Result<TaskRef>
       where A: TArg, R: TRet, F: TFunc<A, R> {
58   let stack = alloc_stack(stack_size)?;
59   let mut new_task = Task::new(task_name, stack, ...)?;
60   let trampoline_offset = new_task.stack.size_in_bytes() -
         size_of::<usize>() - size_of::<RegisterCtx>();
61   let initial_context: &mut RegisterCtx = new_task.stack
         .as_type_mut(trampoline_offset)?;
62   *initial_context = RegisterCtx::new(task_wrapper::<F,A,R>);
63   new_task.saved_stack_ptr = initial_context as *const RegisterCtx;
64   let func_arg: &mut Option<(F, A)> = new_task.stack.as_type_mut(0)?;
65   *func_arg = Some((func, arg));
66   Ok(TaskRef::new(new_task))
67 }
68 fn task_wrapper<F,A,R>() -> ! where A: TArg, R: TRet, F: TFunc<A,R> {
69   let opt: &mut Option<(F, A)> = current_task.stack
         .as_type(0).unwrap();
70   let (func, arg) = opt.take().unwrap();
71   let res: Result<R, KillReason> = catch_unwind_with_arg(func, arg);
72   match res {
73     Ok(exit_value)   => task_cleanup_success::<F,A,R>(exit_value),
74     Err(kill_reason) => task_cleanup_failure::<F,A,R>(kill_reason),
75   }
76 }
77 fn task_cleanup_success<F,A,R>(exit_value: R) -> !
       where A: TArg, R: TRet, F: TFunc<A, R> {
78   current_task.set_as_exited(exit_value);
79   task_cleanup_final::<F,A,R>()
80 }
81 fn task_cleanup_failure<F,A,R>(kill_reason: KillReason) -> !
       where A: TArg, R: TRet, F: TFunc<A, R> {
82   current_task.set_as_killed(kill_reason);
83   task_cleanup_final::<F,A,R>()
84 }
85 fn task_cleanup_final<F,A,R>(curr_task: TaskRef) -> !
       where A: TArg, R: TRet, F: TFunc<A, R> {
86   runqueue::remove_task(current_task());
87   scheduler::schedule(); // task is descheduled, will never run again
88   loop { }
89 }
```

Listing 3: The interface to spawn a task (L57) creates a new task and sets up its stack such that it will jump to `task_wrapper()` upon first context switch, which will then invoke its entry function normally. Every function that handles a task lifecycle stage is parameterized with the same set of trait bounds (L54-56), ensuring that a task's type information (function, argument, return type) is losslessly preserved across its entire lifecycle. Code simplified for brevity.

cycle, as shown in Listing 3: spawning and entering new tasks (L57,68), modifying task runstates as they run, and exiting and cleaning up tasks (L77,81,85). Theseus enforces the following invariants to empower the compiler to uphold memory safety and prevent resource leaks throughout the task lifecycle.

T.1: *Spawning a new task must not violate memory safety.* Rust already ensures this for multiple concurrent userspace threads, as long as they were created using its standard library thread type. Instead of using the standard library, Theseus provides its own task abstraction, overcoming the standard library's need to extralingually accommodate unsafe, platform-specific thread interfaces, e.g. `fork()`. Theseus does not offer `fork` because it is known to be unsafe and unsuitable for SAS systems [7], as it extralingually duplicates task context, states, and underlying memory regions without reflecting that aliasing at the language level.

Theseus's task abstraction preserves safety similarly to and as an extension of Rust threads. The `spawn_task()` interface (L57) requires specifying the exact type of the entry

function F, argument A, and return type R, with the following constraints: (*i*) the entry function must be runnable only once (FnOnce in L54), (*ii*) the argument and return type must be safe to transfer between threads (Send in L55-56), and (*iii*) the lifetime of said three types must outlast the duration of the task itself. All task lifecycle functions are lossless and have identical type parameters (F,A,R), allowing the compiler to naturally extend its safety guarantees to concurrent execution across multiple Theseus tasks and to statically prevent invalidly-typed task entry functions, arguments, and return values.

T.2: *All task states must be released in all possible execution paths.* Releasing task states requires special consideration beyond simply dropping a Task object to prevent resource leakage (§4.2). Task states such as the stack are used during unwinding and can only be cleaned up once unwinding is complete, and task cleanup comprises multiple stages that each permit varying levels of resource release. For example, a task's stack and saved register context can be released when it is exited (L78) or killed (L82), but its runstate and exit value must persist until it has been reaped (not shown).

In addition, there exist multiple potential paths in the end stages of the task lifecycle that each require different cleanup actions. When a task runs to completion, its entry function naturally returns execution to the task_wrapper (L73), which can then safely mark the task as exited with its exit value. When a task crashes, the exception handler starts the unwinding procedure to release all task-held resources, after which it invokes the task failure function (L81) that marks the task as crashed. Both normal and exceptional execution paths invoke a final task cleanup function (L85) that removes the task from runqueues and deschedules it. All of these functions are parameterized with <F,A,R> types, a key part of intralingual fault recovery mechanisms like restartable tasks (§6.2).

T.3: *All memory transitively reachable from a task's entry function must outlive that task.* Although all memory regions in Theseus are represented by MappedPages, which prevents use-after-free via lifetime invariants, it is difficult to use Rust lifetimes to sufficiently express the relationship between a task and arbitrary memory regions it accesses. This is because a Rust program running as a task cannot specify in its code that its variables bound to objects in memory are tied to the lifetime of an underlying MappedPages instance, as they are hidden beneath abstractions like stacks, heaps, or program sections. Even if possible, this would be highly unergonomic and inconvenient, rendering ownership useless. For example, all local stack variables would need to be defined as borrowed references with lifetimes derived from that of the MappedPages representing the stack.

Thus, to uphold this invariant, we instead establish a chain of ownership: each task owns the cell that contains its entry function, and that cell owns any cells it depends on, given by the per-section dependencies in the cell metadata (§3.1). As such, the MappedPages regions containing all functions and data reachable from a task's entry function are guaranteed to outlive that task itself. This avoids littering lifetime constraints across all program variables, and allows Rust code to be written normally with the standard assumption that the stack, heap, data, and text sections will always exist.

In contrast, conventional task management leaves the enforcement of these invariants to the OS programmer, an extralingual approach. In Theseus, only swapping stack pointer registers during a context switch is not intralingual.

5 State Management in Theseus

The third design principle Theseus follows is to minimize and ideally eliminate state spill in its cells. As Theseus's component structure is based on cells, state spill can only occur in interactions (e.g., function calls) that cross a cell boundary and result in changed state(s) in the receiving cell.

5.1 Opaque Exportation through Intralinguality

Theseus employs *opaque exportation* to avoid state spill in client-server interactions: each client is responsible for owning the state that represents its progress with the server, hence *exportation*, but cannot arbitrarily introspect into or modify that server-private state due to type safety, hence *opaque*. Opaque exportation is only possible because Theseus's safe, intralingual design enables shifting the burden of resource/progress bookkeeping from the OS into the compiler. This allows bookkeeping states to be *distributed*, or offloaded to each client, e.g., held only on a client task's stack. Theseus's unwinder can still find and invoke cleanup routines without needing OS knowledge about which resources a client has acquired, thus the server and OS at large need not maintain bookkeeping states for each client.

Conversely, Theseus eschews traditional state encapsulation, in which a server holds all states representing its clients' progress and resource usage [16, 17]. Such encapsulation constitutes state spill and causes fate sharing that breaks isolation: when a server crashes and loses its state, its clients will also fail. Opaque exportation still preserves *information hiding* [52], a primary benefit of encapsulation.

A corollary of opaque exportation is *stateless communication* (à la RESTful web architectures [24]), which dictates that everything necessary for a given request to be handled should be included in that request. Servers that employ stateless communication need not store intermediary states between successive client interactions, as future interactions will be self-sufficient, containing previously-exported states.

Opaque exportation enables Theseus to avoid common spillful abstractions such as handles. Client-side handles to server-owned data forces the server to maintain a global table that associates each client's handle with its underlying resource object, a form of state spill. Theseus rejects handles in favor of a client directly owning the underlying resource object; for example, an application task owns a MappedPages object instead of a virtual address handle, as shown by mp_pgs in L4 of Listing 1. This relieves the server (mm cell) from

the burden of maintaining a handle table, e.g., a list of virtual memory areas (VMAs) that correspond to the virtual addresses given to clients as handles for mapped regions. Note that clients are only responsible for owning, not cleaning up, objects that represent resources they acquired; when said object falls out of scope (or during unwinding), it is cleaned up via compile-time insertion of a server-provided cleanup routine, i.e., the object's drop handler. Thus, Theseus decouples the duty of owning and holding a state from the responsibility of implementing and invoking its cleanup functionality.

Accommodating Multi-Client States: Server-defined resources may pertain to or be shared across more than one client. Thus, Theseus extends opaque exportation to enable all pertinent clients to jointly own that resource state, i.e., *multi-client states*. Joint ownership and resource sharing in general can be realized via heap-allocated objects with automatic reference counting (e.g., `Arc`); while this can be viewed as state spill into the heap, considering spill into the allocator itself is not useful for two reasons. First, heap allocations are represented by owned objects elsewhere that point back to the heap, e.g., types like `Box` or `Arc`. Therefore, it suffices to consider only the propagation of those owned objects when determining where state spill occurred, rather than observing the internal state of the heap itself. Second, state spill into the heap is unavoidable; every basic action from creating a new local string variable to invoking a function would constitute state spill into the heap or stack, rendering it a useless metric.

5.2 Management of Special States in Theseus

Theseus cells often hold *soft states*, those that can be lost or discarded without error [19, 55]. Soft states exist for the sake of convenience or performance, e.g., an in-memory cache of a clock source's period read from hardware. Although soft states technically constitute state spill, they can be idempotently re-obtained or recalculated with no impact on correctness. Therefore, Theseus permits soft states as harmless state spill with no adverse effects on evolution or availability.

We identify *unavoidable states* in two general forms: (*i*) *clientless* states, those that hardware requires the OS to maintain on its behalf, and (*ii*) states needed to handle asynchronous, hardware-invoked entry points that do not provide sufficient context. The former renders opaque exportation impossible and the latter violates stateless communication. In the first case, we cannot modify the behavior or capacity of underlying hardware to accommodate exported states. Thus, Theseus must hold these states to ensure they persist throughout all execution. Examples include low-level x86 structures like the Global Descriptor Table (GDT), Task State Segment (TSS), Interrupt Descriptor Table (IDT). In the second case, Theseus must store necessary contextual states with a static lifetime and scope that exceeds that of the asynchronous hardware event's entry function, e.g., an interrupt handler.

To preserve the interchangeability of server cells in both such cases, Theseus assigns their states a well-defined owner and static lifetime by moving them into `state_db`, a state storage facility with minimal semantics akin to key value databases. Any singleton cell can move its static state into `state_db` and get a weak reference in return, a form of soft state. The `state_db` retains interchangeability despite harboring states spilled from other cells, as it uniquely must cooperate in its own swapping process by hardening itself via serialization to nonvolatile storage. The only other similar cell is the cell manager, which must also serialize its cell metadata. This design decouples a hardware state's lifetime from that of the server cell interacting with it, enabling said cell to be evolved without losing mandatory system-wide states.

5.3 Intralinguality and Spill Freedom: Examples

We further illustrate the relationship between intralingual design and state spill freedom with two example subsystems: memory and task management.

Memory Management: Theseus's `MappedPages` type (§4.3) eliminates state spill through opaque exportation: the client requesting the mapping owns the resultant `MappedPages` object, e.g., `mp_pgs` on L4, rather than the server (`mm` cell) that created it. In contrast, `mm` entities in existing OSes harbor state spill in the form of metadata representing each memory mapping, e.g., a list or table of virtual memory area (VMA) objects; clients must blindly trust that the underlying mapping and VMA persist throughout the usage of their virtual address handle. Importantly, we consider page tables to be hardware-required MMU states, much like x86's GDT or TSS. Page table entries are not language-level objects with lasting variable name bindings in Theseus; thus, writing to a page table is a hardware-externalized side effect rather than state spill. Crucially, the state representing this side effect — the transition from "unmapped" to "mapped" — is not lost, but reflected in the client-side `MappedPages` object rather than a hidden server-side state change.

Task Management: Theseus's intralingual design and its ensuing opaque exportation *significantly* reduce the scope and size of its `Task` struct, thus avoiding most instances of state spill from other subsystems into its task management cells. This is possible because the unwinder and compiler together retain the ability to fully clean up a task's acquired resources, even those shared across tasks, without needing to consult its task structure for resource bookkeeping states. Theseus also moves task-related states specific to other OS features, e.g., runqueue and scheduler information, out of the task struct and into those components themselves. This better follows separation of concerns than conventional OSes that hoard a huge list of OS states needed for manual resource bookkeeping and task cleanup into a centralized, all-encompassing task struct. Such a task struct design causes myriad OS operations to spill state into the task management entities and results in cross-cutting dependencies that closely entangle entities together, hindering their evolution or recovery. Thus, Theseus's task struct can contain only the bare necessities, e.g., the task's

runstate, stack, and saved execution context (register values). Correspondingly, it excludes lists of open files, open sockets, memory mappings, wait queues, etc.

6 Realizing Evolvability and Availability

To demonstrate the utility of Theseus's design, we implement mechanisms inside it to realize challenging computing goals: live evolution and fault recovery.

6.1 Live Evolution via Cell Swapping

The fundamental evolutionary mechanism in Theseus is *cell swapping*, a multi-stage procedure that replaces O "old" existing cells with N "new" ones; O need not equal N. (*i*) First, Theseus loads all new cells into a new empty CellNamespace (§3.1), an isolated linking environment. (*ii*) Theseus then verifies dependencies bidirectionally: new cells must satisfy existing dependencies fulfilled by the old cells, and existing cells must satisfy the new cells' dependencies. Isolated loading allows this to occur before making invasive changes to the running system. (*iii*) Theseus redirects all cells that depend on the old cells to depend on the corresponding new cells, which involves rewriting their relocation entries and dependency metadata, updating on-stack references to the old cells, and transferring states if necessary. (*iv*) Finally, Theseus atomically removes the old cells and symbols from the CellNamespace whilst moving in the new cells.

Evolving a running instance of Theseus is as easy as committing to its repository, which triggers our build server tool to re-compile Theseus and generate an evolution manifest file specifying which new cells shall replace which old ones. Maintainers can also select individual cells to evolve, and all others that must be evolved alongside them are automatically included to ensure a well-formed evolution manifest.

Theseus's design facilitates cell swapping and simplifies known live update techniques like quiescence and state transfer. In stage (*i*), runtime cell bounds let Theseus's dynamic loader ensure that a cell's sections will not overlap or be interleaved in memory with those of another, allowing each cell to claim sole ownership of its memory regions and be cleanly removable in stage (*iv*). Dynamic loading also produces precise dependency information, needed in stages (*ii*) and (*iii*).

Spill-free design of cells in Theseus simplifies state transfer. As previously mentioned, opaque exportation allows a server cell to be more easily swapped because it need not maintain state between successive interactions with clients, increasing its quiescent periods. Stateless communication reduces a given function's dependencies on other cells because it receives necessary states and function callbacks or closures via its arguments. Overall, this hastens the dependency rewriting and state transfer steps in stage (*iii*).

The *cell metadata* accelerates cell swapping. In stage (*ii*), dependency verification amounts to a quick search for fully-qualified symbols in the CellNamespace's symbol map. In stage (*iii*), Theseus need not scan every task's stack, rather only a limited subset for which the old cells' public functions or data are reachable from the task's entry function; reachability is trivially determined by following dependency links in the metadata. Compile-time ownership semantics allow Theseus's cell manager to fearlessly remove old cells and their symbols in stage (*iv*) without first checking for their usage elsewhere, as the compiler has already ensured a removed old cell will not be actually dropped and unloaded until it is no longer referred to by any other cells; this avoids a computationally-complex graph traversal over all metadata.

Theseus's intralingual design extends to *transfer functions* needed for evolving a data structure in stage (*iii*). We allow and require such functions to be implemented intralingually using Rust's type conversion traits, e.g., `Into`. Generation of transfer functions is ongoing work, thus the results reported in §7.1 use manually-implemented transfer functions.

6.2 Availability via Fault Recovery

We next describe how Theseus recovers from language-level exceptions (Rust panics) and hardware-induced faults like CPU exceptions. Theseus follows a multi-stage, cascading approach towards fault recovery, taking increasingly drastic measures until normal execution is recovered. A system-wide fault log records fault context (e.g., instruction pointer, current task) and the recovery action taken in order to track progression through recovery stages and avoid recurring fault loops.

The first recovery stage is to simply tolerate the fault by fully cleaning up a failed task via unwinding. This form of fault isolation allows other tasks that depend on resources shared with the failed task to continue running.

The second recovery stage is to respawn a new instance of the failed task. We extend the existing task infrastructure (Listing 3) to provide a fully intralingual implementation of *restartable tasks*, in which the spawn interface further constrains the `<F,A,R>` type parameters to enable the compiler to check that tasks are well-formed and safely restartable. The augmented trait bounds are `F: Fn(A) -> R + Clone` and `A: Send + Clone + 'static`, which require that the entry function can be safely executed multiple times (`F: Fn`, not `FnOnce`) and the argument can be safely duplicated (`Clone`).

The most significant recovery stage reuses the cell swapping mechanism (§6.1) to replace corrupted cells with freshly-loaded instances at different memory locations. This approach addresses faults that occur on invalid accesses of cell data or text sections, indicating they have been corrupted (e.g., due to a hardware memory failure). This represents the simplest possible case of cell swapping, with no possibility of missing dependencies or changes to code or data types. Following this, the failed task is restarted (as above), which allows it to successfully execute atop the new cell instance(s).

Notably, Theseus's fault recovery mechanisms operate with few dependencies, allowing it to tolerate faults in the lowest system layers in the face of multiple failed subsystems. The fault-critical TCB of components for each recovery stage are

as follows: (*i*) cleanup of a failed task's states relies upon the unwinder, which only needs a basic execution environment to access the stack and invoke functions; (*ii*) restartable tasks rely upon task spawning; (*iii*) cell replacement relies upon object file parsing, loading, and linking. All of this can safely execute within the context of a CPU exception handler in Theseus. In comparison, fault recovery approaches in reliable microkernels like MINIX 3 [30] require support for context switches, interrupts, IPC, and userspace to work properly.

7 Evaluation

We evaluate Theseus to show that it achieves easy and arbitrary live evolution and increases system availability through fault recovery. We assess the impact of intralingual, state spill-free designs on memory and task management performance and compare Theseus's base performance with that of Linux through a series of benchmarks. All experiments were conducted on an Intel NUC 6i7KYK [2] with 4 (8 SMT) 2.6 GHz cores and 32 GB memory, unless otherwise stated.

7.1 Live Evolution

Theseus's evolutionary mechanisms are implemented in-band, that is, within the OS core and using its own features, which differs from existing works that implement live update functionality out-of-band or on a mature OS. As such, it is difficult to conduct a statistical analysis showing which historical commits can be supported by Theseus's live evolution. Instead, we use case studies (as in Baumann et al. [9]) to demonstrate that Theseus is able to evolve core system components in unique manners beyond prior live update works. Figure 2 shows the time scale of evolutionary stages for three case studies: (a) ITC channels, (b) scheduler and runqueue infrastructure, and (c) an Ethernet driver and network update client.

Inter-Task Communication (ITC) Channels: We show how Theseus can evolve its ITC channel layer (the equivalent of IPC) from an existing synchronous, unbuffered rendezvous channel into a new asynchronous buffered channel. We chose this because the histories of MINIX 3, seL4, and QNX Neutrino reveal significant, necessary evolution in their microkernel cores, most notably the addition of or change from synchronous to asynchronous IPC [4, 22]; all require a standard reboot to apply the change. Here, Theseus advances the state of the art by live evolving (*i*) a fundamental OS primitive that must be implemented within a microkernel, (*ii*) a kernel API that necessitates joint evolution of dependent userspace and kernel entities, and (*iii*) a widely-used component whilst preserving the execution context of those that depend on it.

During this experiment, we spawn multiple applications that exchange messages with each other over multiple synchronous channels, in addition to system tasks (e.g., input event manager) that already use said channel. We then issue a live evolution command at a random point while messages are in flight. Because Theseus can evolve cells independently from execution contexts, it can swap the channel implementation out from underneath a running application without having to kill it. As shown in Figure 2(a), this improves availability by reducing median downtime to 385 μs because it preserves the application's runtime progress, avoiding the domino effect of needing to restart multiple other dependent tasks transitively.

Scheduling and Runqueue Subsystems: In this experiment of Figure 2(b), we replace the existing round-robin scheduler with a new priority scheduler and the existing dequeue-based runqueue with a priority queue. All the while, Theseus runs multiple tasks of varying priorities that print messages, illustrating the visible difference in task execution order and frequency before and after evolution. This showcases Theseus's ability to evolve at runtime the modularity of the OS itself (by changing multiple cell bounds) and core cells used incessantly by many others.

At first glance, this appears trivial because existing OSes can already switch between multiple schedulers at runtime. The key distinction is that Theseus booted as an OS that did not originally contain *a priori* knowledge of or in-band support for multiple schedulers, whereas existing OSes require a scheduler infrastructure with a pre-defined common interface to accommodate multiple scheduler policies. This illustrates a significant benefit: subsystems in Theseus need not incorporate a special design or interface to support multiple versions of a given component, e.g., functions like `schedule()` or `task_switch()` can be unaware of multiple schedulers. Instead, Theseus components can rely upon an arbitrary, out-of-band cell swapping mechanism to evolve or flexibly switch between multiple alternatives, resulting in a simpler design.

Ethernet Driver and Network Update Client: In this experiment of Figure 2(c), we evolve Theseus to fix unreliable network downloads, comprising two cells that must be evolved simultaneously: (*i*) the core Ethernet driver underneath the network stack, and (*ii*) Theseus's evolution client application that sits atop the network stack to communicate with the build update server. This demonstrates Theseus's capacity for coordinated, multi-part evolution (as does the above scheduler case) versus small-scope live updates that only patch one driver function. The new Ethernet driver fixes an insidious bug that caused inconsistent connectivity due to incorrectly setting head and tail registers for the ring buffer of received packets; the new evolution client fixes its HTTP layer usage to properly recover from unexpected remote socket disconnections. We achieve this without losing any NIC configuration settings or packet data progress, tested by downloading files during the evolution and verifying them with checksums.

This case shows that Theseus can provide *availability without redundancy*, e.g., for solitary embedded systems in the field, and better availability atop hardware redundancy, e.g., for datacenter network switches that must be brought down during driver updates. Moreover, it shows that Theseus can perform "meta-evolution," i.e., loading a new evolution client

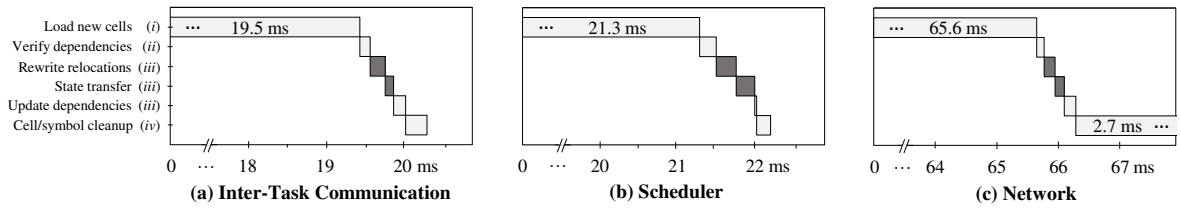


Figure 2: The time taken for each step in Theseus's live evolution procedure, with cell swapping stages marked (*i-iv*) (§6.1). The first two steps are performed in isolation and do not affect the running system. Only the middle two steps (shaded) are critical and may impact execution by requiring atomicity, i.e., a system pause, but this can be avoided when the evolved components robustly handle state unavailability errors, as in the scheduler **(b)** case. The last two steps must lock the cell metadata to prevent overlapping evolution, but do not affect execution.

using that client's own evolutionary features. This procedure is facilitated by state spill freedom that results in network states being owned by the application task, except for minimal states necessary to handle asynchronous receive buffers.

7.2 Fault Recovery

We demonstrate Theseus's ability to tolerate faults within low-level core components, e.g., those that necessarily exist inside a microkernel. We focus on stress-testing whether Theseus can recover from unexpected hardware-induced faults beneath the language, as Theseus can recover from language-level faults easily because the compiler understands and can account for them, guaranteeing that unwinding will work.

Our *fault injection method* is to run Theseus atop the QEMU emulator [11] to enable us to automate arbitrary changes to hardware state, including randomly flipping one bit or overwriting full quadwords in memory, and randomly incrementing the instruction pointer register to skip instructions. We inject faults while running a workload of graphical rendering, task spawning, in-memory FS access, and ITC channel usage, and monitor the workload/OS behavior to determine if and how the fault manifests. This follows common practices in the literature [21, 30, 65]. As found in other fault injection works [30], very few randomly injected faults ($<0.5\%$) manifest into observable failures; thus, we augment our fault injector to target specific regions in memory where faults are likely to manifest, namely a given task's working set of stack, heap, and cell memory (text, data, and rodata sections).

Theseus Recovers from Microkernel-level Faults: Literature on fault-tolerant microkernels, e.g., MINIX 3 [31] and CuriOS [21], only evaluate recovery from faults injected into *userspace* system servers, not the microkernel itself. To show that Theseus supports recovery from faults in such low-level components, we inject faults into both MINIX 3's IPC layer and Theseus's ITC channels and evaluate their ability to recover. To ensure a fair comparison, we manually inspect all layers of MINIX 3's IPC implementation and Theseus's ITC channel implementation to discover 13 faults [45] that cause deterministic failures in both systems.

Out of the 13, Theseus recovers correctly in all but two cases, in which the receiver and sender tasks hang but do not crash; this can be solved via timeouts or resetting the channel. MINIX 3 fails to recover correctly in all 13; its kernel crashes in 11 cases and loses a message in the other two. For example, corrupting the pointer to a passed message that is accessed in the IPC receive routine manifests as an invalid page fault in both Theseus and MINIX 3; MINIX 3's kernel crashes and reboots whereas Theseus unwinds and properly restarts the ITC receiver task, allowing the sender to progress.

General Fault Recovery: To comprehensively assess Theseus's fault recovery, we injected 800,000 faults into subsystems actively used by the above workloads, of which 0.083% manifested as observable failures. Table 1 shows that Theseus successfully recovered from 69% of total manifested faults. Restarting the failed task sufficed in 11% of cases, indicating corrupted stack or heap values; in the remaining 89%, Theseus needed to reload one or more cells, indicating corruption of text or data sections. The observable downtime of Theseus's fault recovery mechanisms is evaluated elsewhere [15].

Theseus failed to recover from 31% of manifested faults, primarily due to the lack of asynchronous unwinding in Rust/LLVM. The compiler generates *synchronous* unwinding tables that only cover instructions where language-level exceptions (Rust panics) may occur. As hardware faults can occur at any instruction, Theseus's unwinder may only find an inexact match for a faulted instruction pointer in the unwinding table, with a cleanup routine that may not completely release all resources acquired at the point of failure. Note that only local variables in the excepted stack frame may be missed, all other stack frames are properly handled. Though the known solution of asynchronous unwinding is unsupported, we are exploring OS and compiler solutions to augment coverage of unwinding information, beyond the scope of this work.

In another 30 cases, the fault caused the system or workload task to hang, recoverable via complementary hardware mechanisms like watchdog timers. In the remaining 18 and 62 cases respectively, Theseus failed to reload a new cell to replace the corrupted cell or suffered a fault in the unwinder's code path itself, for which recovery failures are expected. Collectively, these represent the limitations of Theseus's fault recovery.

7.3 Cost of Intralinguality & State Spill Freedom

Though performance is not a primary goal of Theseus, its intralingual and spill-free designs naturally raise performance

Successful Recovery	**461**	
Restart task		50
Reload cell		411
Failed Recovery	**204**	
Incomplete unwinding		94
Hung task		30
Failed cell replacement		18
Unwinder failure		62
Total manifested faults	**665**	

Table 1: Theseus recovers from 69% of manifested faults in our fault injection trials that emulate hardware failures.

questions. In general, we observe and expect a trend in which many spill-free designs incur mild overhead, such as task and heap management, while some perform better, such as `MappedPages`. We compare multiple versions of Theseus with controlled differences to tease out the performance impact of these specific design choices. We also compare against Linux by porting LMBench microbenchmarks to Rust and running them on both Linux and Theseus; as Theseus is experimental and lacks POSIX support, results should be regarded as *informative* rather than conclusive. Overall, we do not observe any glaring performance penalties herein.

`MappedPages`: Better Performance and Scalability: Figure 3 compares our `MappedPages` design with a conventional spill-ful memory mapping implementation that encapsulates a red-black tree of VMAs, carefully modeled after and optimized to match Linux's behavior. `MappedPages` performs slightly better because (*i*) clients directly own `MappedPages` objects, a form of distributed bookkeeping that obviates the need to search the VMA tree for the memory region that contains a given virtual address, and (*ii*) memory safety invariants are upheld at compile-time. Overall, this difference is unlikely to significantly impact real system workloads.

Avoiding Task State Spill has Negligible Overhead: As described in §5.3, Theseus eliminates runqueue and scheduler states spilled into the task struct, subverting the conventional all-inclusive task struct. This imposes the overhead of iterating through and removing a dead task from *all* runqueues rather than just the runqueue(s) it is known to be on. We evaluate the worst case in which a task is known to be on only *one* runqueue; the more runqueues a task is on, the less relative overhead Theseus has. We run Theseus on a 36-core (72 SMT) Supermicro 119u-7 server with one runqueue per hardware thread, to accurately reflect caching effects when searching through runqueues on other cores. The experiment of Figure 4(a) repeatedly removes a non-running task from its runqueue; while this is a contrived scenario impossible in any OS workload, it does show that overhead increases with the number of runqueues. The experiment of Figure 4(b) spawns and runs a dummy task that immediately exits, measuring the worst possible *realistic* overhead. Here, the impact is negligible because the prerequisite of spawning a task dominates the overhead of removing it from every runqueue.

Heap Designs	*threadtest*	*shbench*
unsafe	20.27 ± 0.009 s	3.99 ± 0.001 s
partially-safe	20.52 ± 0.010 s	4.54 ± 0.002 s
safe	24.82 ± 0.006 s	4.89 ± 0.002 s

Table 2: Heap microbenchmark results for various design points. Threadtest [12] allocates and deallocates 100 million 8-byte objects; shbench [34] does so for 20 million objects of size 1 to 1000 bytes.

Intralingual Heap Bookkeeping causes Overhead: Table 2 shows that an intralingual, safe heap implementation can impose up to 22.5% overhead in bookkeeping costs over an unsafe version. Each heap design variant is based on Theseus's slab [14] allocator that tracks available memory as lists of `MappedPages`, one per slab, which serves allocation requests of a specific size. Multiple heap instances exist within a single `alloc`/`dealloc` interface, matching Rust's language model (§4.1). The unsafe heap design maintains raw pointers to allocation metadata and neither owns its backing `MappedPages` nor knows of their lifetimes. The partially-safe heap owns its backing `MappedPages` but embeds raw pointers to them within the allocation metadata, discarding lifetime information. The safe heap maintains a collections type (e.g., red-black tree) that maps a virtual address to its allocation metadata and its backing `MappedPages`, allowing the compiler to observe and check that the association between an allocation and its backing `MappedPages` is never lost. This is crucial for Theseus to safely exchange memory between multiple per-core heaps, but causes overhead during deallocation when looking up the allocation metadata for a given address.

Microbenchmark Comparisons with Linux: We reimplement select LMBench benchmarks [47] in safe Rust on both Linux and Theseus, omitting those irrelevant to core OS components or with no equivalent in Theseus (e.g., RNG latency, futexes), and those that test subsystems still rudimentary in Theseus (e.g., networking, filesystems). Table 3 shows the results of each benchmark as the mean value across 100,000 iterations; full details are available elsewhere [15]. We do not claim that Theseus generally outperforms existing OSes like Linux, as larger-scale workloads may reveal different trends, but our results do not indicate significant performance drawbacks. The differences shown stem from eliminating the overhead of switching between hardware protection modes and address spaces; these are known benefits of SAS/SPL OSes [33].

In addition, we compare against microkernel IPC fastpaths by implementing an ITC fastpath within Theseus that bypasses the disconnection semantics of Theseus's channels. We realize this fastpath in fully safe code via shared references to an atomic type that holds a small message, achieving a 1-byte RTT of 687 cycles compared to seL4's [41] *one-way* IPC fastpath latency of 401 cycles on the same hardware (without Meltdown mitigations). For reference, Theseus's asynchronous channel has an RTT of 1664 cycles, close to Singularity's reported 1415-cycle channel RTT [5].

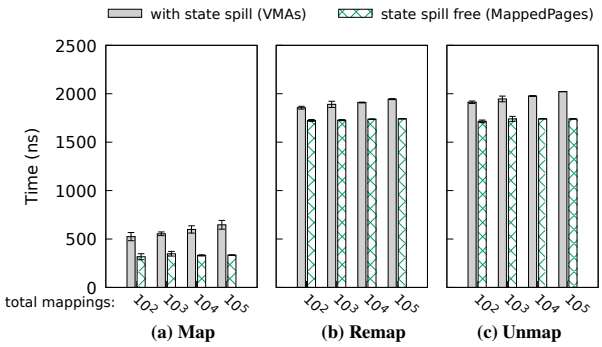


Figure 3: The time to map, remap, and unmap a 4 KiB page is constant for Theseus's spill-free `MappedPages` approach, slightly better than a traditional spillful approach based on a red-black VMA tree.

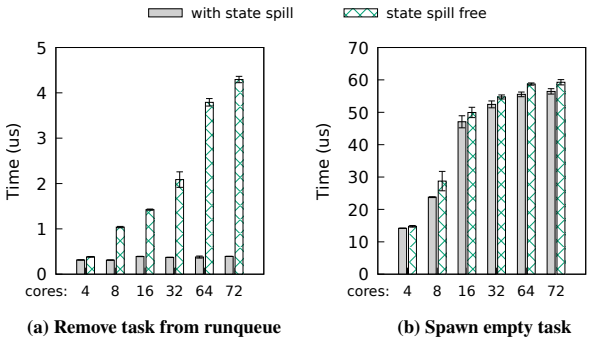


Figure 4: (a) The time to remove a task from the runqueue(s) increases when eliminating runqueue states from the task struct, but is minor in the worst realistic case of (b) spawning an empty task.

LMBench Benchmark	Ported behavior on Theseus; (Linux behavior, if different)	*Linux (Rust)*	*Theseus*	*Theseus (static)*
null syscall	call `curr_task()` function; (invoke `getpid()` syscall in vDSO)	0.28 ± 0.01	0.02 ± 0.00	0.02 ± 0.00
context switch	switch between two threads that continuously yield	0.61 ± 0.06	0.35 ± 0.00	0.34 ± 0.00
create process	spawn "Hello, World!" application; (fork + exec)	567.78 ± 40.4	242.11 ± 0.88	244.35 ± 0.06
memory map	map, write, then unmap 4KiB page; (use `MAP_POPULATE` flag)	2.04 ± 0.15	1.02 ± 0.00	0.99 ± 0.00
IPC	1-byte RTT over async ITC; (non-blocking pipe between threads)	3.65 ± 0.35	1.06 ± 0.00	1.03 ± 0.00

Table 3: Microbenchmark results in microseconds, smaller is better. *Linux (Rust)* is LMBench benchmarks reimplemented in safe Rust on Linux, *Theseus* is those benchmarks on Theseus, and *Theseus (static)* is those benchmarks on a statically-linked build of Theseus. Standard deviations of zero indicate values smaller than the timer period of 42 ns, and cannot be accurately measured.

Finally, the rightmost column of Table 3 shows that the *overhead of runtime-linked code* due to dynamic cell loading in Theseus is generally negligible. For this, we run the same set of benchmarks atop a build of Theseus in which all kernel cells are statically linked into a monolithic kernel binary.

8 Limitations and Discussion

Unsafe Code is an Unfortunate Necessity in a low-level kernel environment, needed to interface with hardware because the compiler understandably lacks a model of hardware semantics. Not all unsafe code is equal; we distinguish between two types of unsafe code: *innocuous* and *infectious*, in which infectious code may violate isolation but innocuous cannot. Unsafe code is infectious if it can circumvent the type system to access data inside another component, thereby "infecting" it, e.g., by dereferencing arbitrary pointers, but is *innocuous* if it merely accesses data reachable from safe code, e.g., writing the address of a variable to an I/O port. Innocuous code can still cause incorrect behavior. As part of ongoing work, we develop a compiler plugin to automate checks for the reachability and type safety of addresses accessed in unsafe blocks; this currently supports language-level unsafe blocks, e.g., within `MappedPages`, but requires manual whitelisting of inline assembly, e.g., context switch routines.

Reliance on Safe Language: Theseus must trust the Rust compiler and its `core`/`alloc` libraries to uphold safety without soundness holes. Fortunately, the risk of trusting Rust is continually decreasing as multiple ongoing works strive to improve and verify the Rust compiler and its base libraries by checking unsafe usage [36, 37, 53]. To enjoy Theseus's benefits, components must be implemented in safe Rust. Legacy code in other safe or managed languages could be supported by implementing their VMs/runtimes in Rust, but unsafe languages require hardware protection or dynamic interposition on memory accesses (à la SFI [60]) if isolation was desired.

Spillful Abstractions: There is tension between achieving spill freedom and supporting existing abstractions that need or benefit from state spill. One example is each task's runnability state that represents whether it is blocked. Choosing a fully spill-free implementation would remove that state altogether, resulting in wasted CPU cycles as tasks would have no alternative but to endlessly spin while waiting on unavailable resources (e.g., acquired locks). As this impacts performance and convenience, we resolve the tension by seeking the middle ground: a minimal boolean state is spilled into each task that represents its runnability. This avoids the high cost of fruitlessly spinning but stops short of a traditional, fully-spillful design that spills information into the task struct about who blocked it and why (the conditions). In Theseus, the state of that blocked condition exists in and is owned by each entity that blocked that task, as per intralingual design.

Similar tensions exist in other subsystems, such as filesystems (FS) and memory. A fully spill-free FS would break existing POSIX interfaces by granting sole ownership of a file

to the client currently accessing it, meaning that the file would appear to be absent until the client releases it. We have not yet deeply explored custom filesystems, so Theseus's current tradeoff is to support legacy FS standards at the cost of accepting state spill in the FS cells. For memory mapping, Theseus fully embraces the spill-free design choice, `MappedPages`.

Limitations of Intralinguality go beyond the overhead imposed by select designs (§7.3) or runtime bounds checks [20]. First, integrating existing unsafe components or libraries into the system can break the chain of compiler knowledge, i.e., intermixed extralinguality limits the benefits of other intralingual components. Second, not all knowledge is available statically; runtime checks may be necessary for nondeterministic input, such as user-specified memory mapping flags. Third, additional design effort is needed to express invariants using the type system versus using simple runtime checks, though this quickly becomes advantageous in OS contexts with complex runtime conditions that are tricky to get right.

9 Related Work

Theseus draws inspiration from much prior work. Related to its use of *safe language*, i.e., Rust, numerous prior works use safe languages in OSes: Modula-3 in SPIN [13], Java in JX [29], C# in Singularity [33], Rust in Tock [44] and Redox [3], and Go in Biscuit [20]. Many recent works have specifically leveraged Rust's safety to realize efficient isolation [42, 49, 51, 66]. Theseus's intralingual design approach (§4.2) goes beyond using a safe language, empowering the compiler to subsume resource-specific invariants into existing ones and thus check safety and correctness to a greater extent.

Theseus's use of a *single address space* and *single privilege level* was inspired by SPIN [13] and Singularity [33], but for a different purpose than performance: matching the OS's runtime model to that of the language (§4.1).

Our work is motivated by recently diagnosed problems in systems software due to *state spill* [16] and the ensuing argument for a spill-free OS [17]. Other works have implicitly targeted symptoms of state spill, e.g., CuriOS [21] shows that holding client-relevant states in server processes complicates fault recovery. CuriOS moves said states into each client's address space, temporarily mapping them into a given server's address space during an interaction; this offers effective isolation but incurs overhead, and only works for userspace servers in a microkernel OS. Theseus isolates client and server states within the same SAS and SPL using type and memory safety.

Theseus employs dynamic loading for *runtime-persistent bounds* of its cells. Dynamic loading is common in OSes to support kernel extensibility [13, 50, 56, 64], but only to load new modules, such as drivers and extensions, alongside (not in place of) a large, monolithic kernel without clear runtime bounds. Jacobsen et al. embed a microkernel within the Linux kernel as an indirection layer to decompose Linux into lightweight capability domains [35]; this helps to isolate kernel subsystems but not to evolve or recover them.

Microkernel OSes [21, 31, 41] have persistent bounds for OS services that run in hardware-isolated userspace processes. Genode [23] is a similarly-modularized OS framework that creates a hierarchical tree of processes for strong access control. These OS structures make it easier to recover from service failures or update an OS service by restarting its process, but modularizing along coarse-grained process bounds limits their ability to evolve and recover from faults in core microkernel components. Also, Theseus's finer-grained components make hardware-enforced process bounds uneconomical.

Live update of systems software has been extensively studied. Many works retrofit live update into legacy OSes like Linux [6, 18, 39, 46, 54, 58, 63]. Existing solutions need deep kernel expertise or tedious manual effort to generate or apply an update [18, 46, 54]; some impose overhead due to intermediary layers of indirection [18, 32, 57] or full-system checkpointing [39]; others are unable change kernel APIs, internal data structures, or non-function entities [6, 54, 58]. Overall, these works target small, localized security patches. In contrast, Theseus can apply sweeping evolutionary changes to core kernel components, their modularity, and kernel APIs by virtue of its new OS structure and spill-free design.

K42 [9, 10, 32, 57] is an object-oriented OS that deeply explores live update via hot-swapping of objects, similar to Theseus's cell swapping. Unlike Theseus, K42 requires a uniform indirection layer atop all objects and can swap *only* objects, not low-level code beneath the OOP language layer, e.g., exception handling or hardware interaction. Similarly, microkernel solutions like PROTEOS [28], based on MINIX 3, can accommodate complex system updates for userspace server processes. Theseus builds upon PROTEOS's novel techniques for state transfer, but can evolve finer-grained components, including those within a microkernel.

Fault-tolerant OS literature spans a wide variety of approaches, including using software domains to isolate and recover from failures in drivers and select OS subsystems [43, 59, 60], hardware isolation between OS service processes in microkernels [21, 31], and checkpoint/restore of drivers [38] or OS services [30] for faster, stateful recovery. Theseus uses intralingual mechanisms like unwinding and restartable tasks to ensure that language-level safety assumptions and compiler-provided isolation are not violated by recovery actions. Theseus also distinguishes between recovering a component (cell) and an execution context (task), can recover and replace finer-grained components than processes, and leverages novel state management techniques to simplify recovery logic.

Acknowledgments

This work is supported in part by NSF Awards #1422312, #2016422, and their REU supplements. We are grateful to the anonymous reviewers and our shepherd Malte Schwarzkopf, whose input strengthened our final paper.

References

[1] The DWARF debugging standard. http://dwarfstd.org/. Accessed: 2020-05-08.

[2] Intel NUC Kit NUC6i7KYK technical specifications. https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc6i7kyk.html. Accessed: 2020-04-26.

[3] Redox - your next(gen) os. https://www.redox-os.org/. Accessed: 2017-08-11.

[4] The QNX Neutrino Microkernel – QNX Neutrino IPC. http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html#NTOIPC. Accessed: 2020-05-22.

[5] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proc. ACM Workshop on Memory System Performance and Correctness*, 2006.

[6] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. ACM EuroSys*, 2009.

[7] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proc. HotOS*, 2019.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proc. ACM SOSP*, 2009.

[9] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proc. USENIX ATC*, 2005.

[10] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *6th Linux. Conf. Au*, 2005.

[11] Fabrice Bellard. QEMU: a fast and portable dynamic translator. In *Proc. USENIX ATC*, 2005.

[12] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. ACM ASPLOS*, 2000.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM SOSP*, 1995.

[14] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proc. USENIX Summer Technical Conf.*, 1994.

[15] Kevin Boos. *Theseus: Rethinking Operating Systems Structure and State Management*. PhD thesis, Rice University, 2020.

[16] Kevin Boos, Emilio Del Vecchio, and Lin Zhong. A characterization of state spill in modern operating systems. In *Proc. ACM EuroSys*, 2017.

[17] Kevin Boos and Lin Zhong. Theseus: a state spill-free operating system. In *Proc. ACM PLOS*, 2017.

[18] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proc. ACM VEE*, 2006.

[19] David Clark. The design philosophy of the DARPA internet protocols. In *Proc. ACM SIGCOMM*, 1988.

[20] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proc. USENIX OSDI*, 2018.

[21] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. USENIX OSDI*, 2008.

[22] Kevin Elphinstone and Gernot Heiser. From L3 to seL4: What have we learnt in 20 years of L4 microkernels? In *Proc. ACM SOSP*, 2013.

[23] Norman Feske. Genode operating system framework. https://genode.org/documentation/genode-foundations-19-05.pdf, 2015. Accessed: 2017-08-19.

[24] Roy Fielding. Representational state transfer. *Architectural Styles and the Design of Network-based Software Architecture*, 2000.

[25] Glenn Fleishman. In space, no one can hear you kernel panic. https://increment.com/software-architecture/in-space-no-one-can-hear-you-kernel-panic/, February 2020.

[26] U.S. Food and Drug Administration. Firmware update to address cybersecurity vulnerabilities identified in Abbott's (formerly St. Jude Medical's) implantable cardiac pacemakers: FDA safety communication. https://www.fda.gov/medical-devices/safety-communications/firmware-update-address-cybersecurity-vulnerabilities-identified-abbotts-formerly-st-jude-medicals. Published: 2017-08-29.

[27] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. ACM SIGCOMM*, 2011.

[28] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. ACM ASPLOS*, 2013.

[29] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In *Proc. USENIX ATC*, 2002.

[30] Jorrit Herder. *Building a dependable operating system: fault tolerance in MINIX 3*. PhD thesis, Vrije Universiteit Amsterdam, 2010.

[31] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[32] K. Hui, J. Appavoo, R. Wisniewski, M. Auslander, D. Edelsohn, B. Gamsa, O. Krieger, B. Rosenburg, and M. Stumm. Supporting hot-swappable components for system software. In *Proc. HotOS*, 2001.

[33] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 2007.

[34] MicroQuill Inc. Microquill smartheap 4.0 benchmark. http://microquill.com/.

[35] Charles Jacobsen, Muktesh Khole, Sarah Spall, Scotty Bauer, and Anton Burtsev. Lightweight capability domains: Towards decomposing the Linux kernel. *SIGOPS Oper. Syst. Rev.*, 2016.

[36] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for Rust. In *Proc. ACM POPL*, 2020.

[37] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *Proc. ACM POPL*, 2017.

[38] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proc. ACM ASPLOS*, 2013.

[39] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant OS updates via userspace checkpoint-and-restart. In *Proc. USENIX ATC*, 2016.

[40] Steve Klabnik and Carol Nichols. The Rust programming language. https://doc.rust-lang.org/book/. Accessed: 2020-05-22.

[41] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. ACM SOSP*, 2009.

[42] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: bare-metal extensions for multi-tenant low-latency storage. In *Proc. USENIX OSDI*, 2018.

[43] Andrew Lenharth, Vikram S Adve, and Samuel T King. Recovery domains: an organizing principle for recoverable operating systems. In *Proc. ACM ASPLOS*, 2009.

[44] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64KB computer safely and efficiently. In *Proc. ACM SOSP*, 2017.

[45] Namitha Liyanage. Fault recovery in the Theseus operating system. Master's thesis, Rice University, 2020.

[46] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. ACM EuroSys*, 2003.

[47] Larry W McVoy, Carl Staelin, et al. LMbench: Portable tools for performance analysis. In *Proc. USENIX ATC*, 1996.

[48] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proc. ACM IMC*, 2018.

[49] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas Anderson. Practical safe Linux kernel extensibility. In *Proc. HotOS*, 2019.

[50] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. USENIX OSDI*, 1996.

[51] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. USENIX OSDI*, 2016.

[52] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[53] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proc. ACM PLDI*, 2020.

[54] RedHat. Introducing kpatch: Dynamic kernel patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching, 2014.

[55] Angela Schuett, Suchitra Raman, Yatin Chawathe, Steven McCanne, and Randy Katz. A soft-state protocol for accessing multimedia archives. In *Proc. ACM NOSSDAV*, 1998.

[56] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. USENIX OSDI*, 1996.

[57] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W Wisniewski, Dilma Da Silva, Gregory R Ganger, Orran Krieger, Michael Stumm, Marc A Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proc. USENIX ATC*, 2003.

[58] SUSE. SUSE releases kGraft for live patching of Linux kernel. https://www.suse.com/c/news/suse-releases-kgraft-for-live-patching-of-linux-kernel/, 2014.

[59] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proc. USENIX OSDI*, 2004.

[60] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proc. ACM SOSP*, 2003.

[61] Theseus Operating System. https://github.com/theseus-os/Theseus, 2020.

[62] Kim Tingley. The New York Times: The loyal engineers steering NASA's Voyager probes access the universe. https://www.nytimes.com/2017/08/03/magazine/the-loyal-engineers-steering-nasas-voyager-probes-across-the-universe.html, 2017.

[63] Steven J. Vaughan-Nichols. Kernelcare: New no-reboot Linux patching system. https://www.zdnet.com/article/kernelcare-new-no-reboot-linux-patching-system/, 2014.

[64] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proc. ACM SOSP*, 1993.

[65] Long Wang, Zbigniew Kalbarczyk, Weining Gu, and Ravishankar K Iyer. An OS-level framework for providing application-aware reliability. In *Proc. IEEE PRDC*, 2006.

[66] Minhong Yun and Lin Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *Proc. NDSS*, February 2019.

A Artifact Appendix

A.1 Abstract

The full source code and documentation for Theseus OS is available online as a GitHub repository [61], where we invite contributions from the public. All OS components, source artifacts, and experiments described in the paper are present in the repository, along with detailed instructions on how to run or use them.

A.2 Artifact check-list

- **Program:** The Theseus Operating System
- **Compilation:** Rust, Make, no_std, freestanding, bare-metal
- **Binary:** OS .iso images
- **Run-time environment:** x86_64 bare-metal
- **Hardware:** Virtual or real x86_64 machine with BIOS
- **Experiments:** microbenchmarks, live evolution, fault recovery
- **Required disk space:** under 1GB
- **Expected experiment run time:** 10-12 hours
- **Public link:** https://github.com/theseus-os/Theseus/tree/osdi20ae/osdi20ae
- **Code licenses:** MIT

A.3 Description

A.3.1 How to access

The Theseus repository is hosted on GitHub at https://github.com/theseus-os/Theseus. The top level `README` contains detailed instructions on building and running Theseus. The branch `osdi20ae` contains pre-built Theseus images with instructions that specify how to easily reproduce each evaluation experiment, available at https://github.com/theseus-os/Theseus/tree/osdi20ae. The source-level documentation and high-level Theseus book are hosted online at https://theseus-os.github.io/Theseus/, but is best viewed using the commands `make doc` and `make book`, as specified in our `README`.

A.3.2 Hardware dependencies

We have tested Theseus on a variety of real machines, including Intel NUC devices, various Thinkpad laptops, and Supermicro servers. Currently, the only known limiting factor is support for booting via USB or PXE using traditional BIOS rather than UEFI; support for UEFI is a work in progress.

A.3.3 Software dependencies

We have tested building and then running Theseus in QEMU atop of the following host OSes:

- Linux, 64-bit Debian-based distributions like Ubuntu, tested on Ubuntu 16.04, 18.04, 20.04.
- Windows, using the Windows Subsystem for Linux (WSL), tested on the Ubuntu version of WSL and WSL2.
- MacOS, tested on versions High Sierra (10.13) and Catalina (10.15.2).
- Docker container environments.

The specific set of package dependencies are listed in the top-level `README`. Additional packages needed for artifact evaluation only are specified in the `README`s for each experiment.

A.4 Installation

Standard installation procedures are not required; steps to build and run a functional Theseus OS image are listed in our `README`.

A.5 Experiment workflow

All experiments described in the paper are implemented directly within the source code of Theseus and gated by compile-time configuration settings, so they are straightforward to inspect and run. The experiments are divided into the following groups, each of which has an accompanying script and set of instructions describing how to run it within its respective artifact folder in the repository. To further simplify reproduction of results, we provide pre-built OS images that are properly configured for each experimental setup.

- Case studies of live evolution of core OS components
- General fault injection and recovery
- Comparison with IPC fault recovery in MINIX 3
- Overhead of state spill and intralingual designs
  - The cost of `MappedPages` for memory mapping
  - The cost of removing runqueue/scheduler state spill from the task struct
  - The cost of safety and intralinguality in heap allocation
- LMBench microbenchmarks ported to Theseus and Linux

The documentation as well as a pre-built image of Theseus for each experiment can be found in the subfolder with the same name in the `osdi20ae` folder: https://github.com/theseus-os/Theseus/tree/osdi20ae/osdi20ae. Some experiments require Theseus to be built with special flags or need to be passed certain parameters to match the test cases in the paper. The requirements for running each benchmark as it was run in the paper are given in the documentation.

A.6 Evaluation and expected result

Due to differences in hardware and execution environments, the exact results presented in the paper may differ from those reproduced on other machines. However, the relative performance trends and conclusions drawn in our evaluation should hold.

A.6.1 Live evolution case studies

In these case studies, which correspond to §7.1 and Figure 2 in the paper, we start with a standard build of Theseus and evolve it into a different version with completely different functionality. This process downloads a set of modified crates as specified by an evolution manifest generated by our build tool, and then applies them to the running system using the `upd` application. The experiments can either be reproduced using a host machine that runs our build server alongside a virtualized instance of Theseus within QEMU, or using two separate physical machines with network access, one for the build server and one for Theseus. We provide detailed instructions on how to set up and reproduce each case study, as well as screenshots that describe what new behavior is expected after the evolutionary procedure has completed. We also explain how to obtain the raw values and calculate the measurements in Figure 2; one expects a general trend in which the first step of loading and linking crate object files takes the longest, and the critical third and fourth steps are very fast, within tens of μs to a few hundred μs.

A.6.2 Fault injection and recovery

In this experiment, Theseus runs atop the QEMU emulator, and we attach GDB to the virtualized instance of Theseus and use a script to inject faults into it by modifying the contents of memory and other hardware components like the instruction pointer register. We provide pre-built images of Theseus that run two of the sample workloads described in §7.2: accessing an in-memory filesystem and using inter-task communication channels. Each image is accompanied by a script that will inject faults into the system components used by the workloads run in that image. At the end of the experiment, each script should output the number of successful recoveries and failed recoveries. We also provide a full CSV table listing every fault injected and its outcome in our own trials.

A.6.3 IPC fault comparison

In this experiment, we compare 13 deterministic faults injected into Theseus's ITC channels and MINIX 3's IPC channels. A table describing the nature of each fault and the expected response observed in both Theseus and MINIX 3 is given in the `README` in this experiment's folder. As described in that `README`, modified source code of MINIX 3 is available at https://github.com/theseus-os/minix_osdi_ae, which contains a separate branch for each fault and instructions on how to build and run both systems to reproduce said fault recovery behavior.

A.6.4 Evaluation of `MappedPages`

This experiment measures the time to map, remap, and unmap a 4KiB page in two configurations: using the state spill-free, intralingual `MappedPages` implementation in Theseus, and using a traditional spillful approach based on a red-black tree of VMAs (see Figure 3). We provide a pre-built image for automated use with QEMU, with an accompanying script that runs this benchmark multiple times, parses the results, and calculates the statistics given in the paper. One should expect to observe similar trends as Figure 3, in which the `MappedPages` approach scales to many concurrent mappings better than the VMA-based approach.

A.6.5 Evaluation of runqueue state spill in tasking

This experiment measures the overhead of eliminating state spill into the tasking subsystem from the runqueue and scheduler subsystems, i.e., the overhead measured in Figure 4. We provide two pre-built images of Theseus (and instructions to re-create them manually), one using our standard spill-free implementation of runqueue and task states and one with a traditional spillful approach of a large stateful task struct. One should expect to observe similar trends as Figure 4(a), in that simply removing an exited task in the spill-free version will scale roughly linearly with the number of total runqueues in the system, whereas the spillful version should remain constant. The more important trend to observe is that of Figure 4(b), in which the overall effect of runqueue-task state spill is relatively minor because the cost of spawning a task dominates that of searching runqueues to remove an exited task.

A.6.6 Heap microbenchmarks

In this experiment, we run the *threadtest* and *shbench* microbenchmarks to measure the performance of three different versions of heap allocators that vary in their levels of safety and intralinguality, as given in Table 2 of the paper. We provide pre-built images for each configuration and instructions on how to build them manually. Overall, the expected trend is that the unsafe heap is the fastest, followed by the partially-safe heap and then the safe heap; the absolute runtimes may change but the relative overhead should remain similar to the paper.

A.6.7 LMBench microbenchmarks

In this experiment, we port a core subset of LMBench benchmarks to safe Rust code and compare their execution times across three environments: as Linux userspace applications, as applications atop the standard dynamically-loaded version of Theseus, and as applications atop a statically-linked version of Theseus, as shown in Table 3. We provide pre-built images for both configurations of Theseus as well as the ported LMBench source code, plus scripts and instructions for building and running it. In these microbenchmarks, we expect Theseus to be generally faster than Linux due to its SAS/SPL design that avoids extra boundary crossings (address spaces and privilege levels) imposed by traditional hardware-protected systems like Linux.

A.7 Experiment customization

The test executables and scripts for each experiment in Theseus can be customized with command-line parameters, e.g., the number of iterations, the size of trial operations, etc. Running each test command with a solitary `--help` argument will output a help menu that describes those parameters, along with in-source documentation at the top of each test application.

A.8 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts

RedLeaf: Isolation and Communication in a Safe Operating System

Vikram Narayanan
University of California, Irvine

Tianjiao Huang
University of California, Irvine

David Detweiler
University of California, Irvine

Dan Appel
University of California, Irvine

Zhaofeng Li
University of California, Irvine

Gerd Zellweger
VMware Research

Anton Burtsev
University of California, Irvine

Abstract

RedLeaf is a new operating system developed from scratch in Rust to explore the impact of language safety on operating system organization. In contrast to commodity systems, RedLeaf does not rely on hardware address spaces for isolation and instead uses only type and memory safety of the Rust language. Departure from costly hardware isolation mechanisms allows us to explore the design space of systems that embrace lightweight fine-grained isolation. We develop a new abstraction of a lightweight language-based isolation domain that provides a unit of information hiding and fault isolation. Domains can be dynamically loaded and cleanly terminated, i.e., errors in one domain do not affect the execution of other domains. Building on RedLeaf isolation mechanisms, we demonstrate the possibility to implement end-to-end zero-copy, fault isolation, and transparent recovery of device drivers. To evaluate the practicality of RedLeaf abstractions, we implement Rv6, a POSIX-subset operating system as a collection of RedLeaf domains. Finally, to demonstrate that Rust and fine-grained isolation are practical—we develop efficient versions of a 10Gbps Intel ixgbe network and NVMe solid-state disk device drivers that match the performance of the fastest DPDK and SPDK equivalents.

1 Introduction

Four decades ago, early operating system designs identified the ability to isolate kernel subsystems as a critical mechanism for increasing the reliability and security of the entire system [12, 32]. Unfortunately, despite many attempts to introduce fine-grained isolation to the kernel, modern systems remain monolithic. Historically, software and hardware mechanisms remain prohibitively expensive for isolation of subsystems with tightest performance budgets. Multiple hardware projects explored the ability to implement fine-grained, low-overhead isolation mechanisms in hardware [84, 89, 90]. However, focusing on performance, modern commodity CPUs provide only basic support for coarse-grained isolation of user applications. Similarly, for decades, overheads of safe languages that can provide fine-grained isolation in software remained prohibitive for low-level operating system code. Traditionally, safe languages require a managed runtime, and specifically, garbage collection, to implement safety. Despite many advances in garbage collection, its overhead is high for systems designed to process millions of requests per second per core (the fastest garbage collected languages experience 20-50% slowdown compared to C on a typical device driver workload [28]).

For decades, breaking the design choice of a monolithic kernel remained impractical. As a result, modern kernels suffer from lack of isolation and its benefits: clean modularity, information hiding, fault isolation, transparent subsystem recovery, and fine-grained access control.

The historical balance of isolation and performance is changing with the development of Rust, arguably, the first practical language that achieves safety without garbage collection [45]. Rust combines an old idea of *linear types* [86] with pragmatic language design. Rust enforces type and memory safety through a restricted ownership model allowing only one unique reference to each live object in memory. This allows statically tracking the lifetime of the object and deallocating it without a garbage collector. The runtime overhead of the language is limited to bounds checking, which in many cases can be concealed by modern superscalar out-of-order CPUs that can predict and execute the correct path around the check [28]. To enable practical non-linear data structures, Rust provides a small set of carefully chosen primitives that allow escaping strict limitations of the linear type system.

Rust is quickly gaining popularity as a tool for development of low-level systems that traditionally were done in C [4, 24, 40, 47, 50, 65]. Low-overhead safety brings a range of immediate security benefits—it is expected, that two-thirds of vulnerabilities caused by low-level programming idioms typical for unsafe languages can be eliminated through the use of a safe language alone [20, 22, 67, 69, 77].

Unfortunately, recent projects mostly use Rust as a drop-in replacement for C. We, however, argue that true benefits of language safety lie in the possibility to enable practical, lightweight, fine-grained isolation and a range of mechanisms

that remained in the focus of systems research but remained impractical for decades: fault isolation [79], transparent device driver recovery [78], safe kernel extensions [13, 75], fine-grained capability-based access control [76], and more.

RedLeaf[1] is a new operating system aimed at exploring the impact of language safety on operating system organization, and specifically the ability to utilize fine-grained isolation and its benefits in the kernel. RedLeaf is implemented from scratch in Rust. It does not rely on hardware mechanisms for isolation and instead uses only type and memory safety of the Rust language.

Despite multiple projects exploring isolation in language-based systems [6, 35, 39, 85] articulating principles of isolation and providing a practical implementation in Rust remains challenging. In general, safe languages provide mechanisms to control access to the fields of individual objects (e.g., through *pub* access modifier in Rust) and protect pointers, i.e., restrict access to the state of the program transitively reachable through visible global variables and explicitly passed arguments. Control over references and communication channels allows isolating the state of the program on function and module boundaries enforcing confidentiality and integrity, and, more generally, constructing a broad range of least-privilege systems through a collection of techniques explored by object-capability languages [59].

Unfortunately, built-in language mechanisms alone are not sufficient for implementing a system that isolates mutually distrusting computations, e.g., an operating system kernel that relies on language safety for isolating applications and kernel subsystems. To protect the execution of the entire system, the kernel needs a mechanism that *isolates faults*, i.e., provides a way to terminate a faulting or misbehaving computation in such a way that it leaves the system in a clean state. Specifically, after the subsystem is terminated the isolation mechanisms should provide a way to 1) deallocate all resources that were in use by the subsystem, 2) preserve the objects that were allocated by the subsystem but then were passed to other subsystems through communication channels, and 3) ensure that all future invocations of the interfaces exposed by the terminated subsystem do not violate safety or block the caller, but instead return an error. Fault isolation is challenging in the face of semantically-rich interfaces encouraged by language-based systems—frequent exchange of references all too often implies that a crash of a single component leaves the entire system in a corrupted state [85].

Over the years the goal to isolate computations in language-based systems came a long way from early single-user, single-language, single-address space designs [9, 14, 19, 25, 34, 55, 71, 80] to ideas of heap isolation [6, 35] and use of linear types to enforce it [39]. Nevertheless, today the principles of language-based isolation are not well understood. Singularity [39], which implemented fault isolation in Sing#, relied on a tight co-design of the language and operating system to implement its isolation mechanisms. Nevertheless, several recent systems suggesting the idea of using Rust for lightweight isolation, e.g., Netbricks [68] and Splinter [47], struggled to articulate the principles of implementing isolation, instead falling back to substituting fault isolation for information hiding already provided by Rust. Similar, Tock, a recent operating system in Rust, supports fault isolation of user processes through traditional hardware mechanisms and a restricted system call interface, but fails to provide fault isolation of its device drivers (capsules) implemented in safe Rust [50].

Our work develops principles and mechanisms of *fault isolation* in a safe language. We introduce an abstraction of a language-based isolation domain that serves as a unit of information hiding, loading, and fault isolation. To encapsulate domain's state and implement fault isolation at domain boundary, we develop the following principles:

- **Heap isolation** We enforce *heap isolation* as an invariant across domains, i.e., domains never hold pointers into private heaps of other domains. Heap isolation is key for termination and unloading of crashing domains, since no other domains hold pointers into the private heap of a crashing domain, it's safe to deallocate the entire heap. To enable cross-domain communication, we introduce a special *shared heap* that allows allocation of objects that can be exchanged between domains.
- **Exchangeable types** To enforce heap isolation, we introduce the idea of *exchangeable types*, i.e., types that can be safely exchanged across domains without leaking pointers to private heaps. Exchangeable types allow us to statically enforce the invariant that objects allocated on the shared heap cannot have pointers into private domain heaps, but can have references to other objects on the shared heap.
- **Ownership tracking** To deallocate resources owned by a crashing domain on the shared heap, we track ownership of all objects on the shared heap. When an object is passed between domains we update its ownership depending on whether it's moved between domains or borrowed in a read-only access. We rely on Rust's *ownership discipline* to enforce that domains lose ownership when they pass a reference to a shared object in a cross-domain function call, i.e., Rust enforces that there are no aliases into the passed object left in the caller domain.
- **Interface validation** To provide extensibility of the system and allow domain authors to define custom interfaces for subsystems they implement while retaining isolation, we validate all cross-domain interfaces enforcing the invariant that interfaces are restricted to exchangeable types and hence preventing them from breaking the heap isolation invariants. We develop an interface definition language (IDL) that statically validates definitions of cross-domain interfaces and generates implementations for them.

[1] Forming in the leaf tissue Rust fungi turn it red.

- **Cross-domain call proxying** We mediate all cross-domain invocations with *invocation proxies*—a layer of trusted code that interposes on all domain's interfaces. Proxies update ownership of objects passed across domains, provide support for unwinding execution of threads from a crashed domain, and protect future invocations of the domain after it is terminated. Our IDL generates implementations of the proxy objects from interface definitions.

The above principles allow us to enable fault-isolation in a practical manner: isolation boundaries introduce minimal overhead even in the face of semantically-rich interfaces. When a domain crashes, we isolate the fault by unwinding execution of all threads that currently execute inside the domain, and deallocate domain's resources without affecting the rest of the system. Subsequent invocations of domain's interfaces return errors, but remain safe and do not trigger panics. All objects allocated by the domain, but returned before the crash, remain alive.

To test these principles we implement RedLeaf as a microkernel system in which a collection of isolated domains implement functionality of the kernel: typical kernel subsystems, POSIX-like interface, device drivers, and user applications. RedLeaf provides typical features of a modern kernel: multi-core support, memory management, dynamic loading of kernel extensions, POSIX-like user processes, and fast device drivers. Building on RedLeaf isolation mechanisms, we demonstrate the possibility to transparently recover crashing device drivers. We implement an idea similar to *shadow drivers* [78], i.e., lightweight shadow domains that mediate access to the device driver and restart it replaying its initialization protocol after the crash.

To evaluate the generality of RedLeaf abstractions, we implement Rv6, a POSIX-subset operating system on top of RedLeaf. Rv6 follows the UNIX V6 specification [53]. Despite being a relatively simple kernel, Rv6 is a good platform that illustrates how ideas of fine-grained, language-based isolation can be applied to modern kernels centered around the POSIX interface. Finally, to demonstrate that Rust and fine-grained isolation introduces a non-prohibitive overhead, we develop efficient versions of 10Gbps Intel Ixgbe network and PCIe-attached solid state-disk NVMe drivers.

We argue that a combination of practical language safety and ownership discipline allows us to enable many classical ideas of operating system research for the first time in an efficient way. RedLeaf is fast, supports fine-grained isolation of kernel subsystems [57, 61, 62, 79], fault isolation [78, 79], implements end-to-end zero-copy communication [39], enables user-level device drivers and kernel bypass [11, 21, 42, 70], and more.

2 Isolation in Language-Based Systems

Isolation has a long history of research in language-based systems that were exploring tradeoffs of enforcing lightweight isolation boundaries through language safety, fine-grained control of pointers, and type systems. Early operating systems applied safe languages for operating system development [9, 14, 19, 25, 34, 55, 71, 80]. These systems implemented an "open" architecture, i.e., a single-user, single-language, single-address space operating system that blurred the boundary between the operating system and the application itself [48]. These systems relied on language safety to protect against accidental errors but did not provide isolation of subsystems or user-applications (modern unikernels take a similar approach [2, 37, 56]).

SPIN was the first to suggest language safety as a mechanism to implement isolation of dynamic kernel extensions [13]. SPIN utilized Modula-3 pointers as capabilities to enforce confidentiality and integrity, but since pointers were exchanged across isolation boundaries it failed to provide fault isolation—a crashing extension left the system in an inconsistent state.

J-Kernel [85] and KaffeOS [6] were the first kernels to point out the problem that language safety alone is not sufficient for enforcing fault isolation and termination of untrusted subsystems. To support termination of isolated domains in Java, J-Kernel developed the idea of mediating accesses to all objects that are shared across domains [85]. J-Kernel introduces a special capability object that wraps the interface of the original object shared across isolated subsystems. To support domain termination, all capabilities created by a crashing domain were revoked hence dropping the reference to the original object that was garbage collected and preventing the future accesses by returning an exception. J-Kernel relied on a custom class loader to validate cross-domain interfaces (i.e., generate remote-invocation proxies at run-time instead of using a static IDL compiler). To enforce isolation, J-Kernel utilized a special calling convention that allowed passing capability references by reference, but required a deep copy for regular unwrapped objects. Without ownership discipline for shared objects, J-Kernel provided a somewhat limited fault isolation model: the moment the domain that created the object crashed all references to the shared objects were revoked, propagating faults into domains that acquired these objects through cross-domain invocations. Moreover, lack of "move" semantics, i.e., the ability to enforce that the caller lost access to the object when it was passed to the callee, implied that isolation required a deep copy of objects which is prohibitive for isolation of modern, high-throughput device drivers.

Instead of mediating accesses to shared objects through capability references, KaffeOS adopts the technique of "write barriers" [88] that validate all pointer assignments throughout the system and hence can enforce a specific pointer discipline [6]. KaffeOS introduced separation of private domain and special shared heaps designated for sharing of objects across domains—explicit separation was critical to perform the write barrier check, i.e., if assigned pointer belonged to a specific heap. Write barriers were used to enforce the follow-

ing invariants: 1) objects on the private heap were allowed to have pointers into objects on the shared heap, but 2) objects on the shared heap were constrained to the same shared heap. On cross-domain invocations, when a reference to a shared object was passed to another domain, the write barrier was used to validate the invariants, and also to create a special pair of objects responsible for reference counting and garbage collecting shared objects. KaffeOS had the following fault isolation model: when the creator of the object terminated, other domains retained access to the object (reference counting ensured that eventually objects were deallocated when all sharers terminated). Unfortunately, while other domains were able to access the objects after their creator crashed, it was not sufficient for clean isolation—shared objects were potentially left in an inconsistent state (e.g., if the crash happened halfway through an object update), thus potentially halting or crashing other domains. Similar to J-Kernel, isolation of objects required a deep copy on a cross-domain invocation. Finally, performance overhead of mediating all pointer updates was high.

Singularity OS introduced a new fault isolation model built around a statically enforced ownership discipline [39]. Similar to KaffeOS, in Singularity applications used isolated private heaps and a special "exchange heap" for shared objects. A pioneering design decision was to enforce *single ownership* of objects allocated on the exchange heap, i.e., only one domain could have a reference to an object on the shared heap at a time. When a reference to an object was passed across domains the ownership of the object was "moved" between domains (an attempt to access the object after passing it to another domain was rejected by the compiler). Singularity developed a collection of novel static analysis and verification techniques enforcing this property statically in a garbage collected Sing# language. Single ownership was key for a clean and practical fault isolation model—crashing domains were not able to affect the rest of the system—not only their private heaps were isolated, but a novel ownership discipline allowed for isolation of the shared heap, i.e., there was no way for a crashing domain to trigger revocation of shared references in other domains, or leave shared objects in an inconsistent state. Moreover, single ownership allowed secure isolation in a zero-copy manner, i.e., the move semantics guaranteed that the sender of an object was losing access to it and hence allowed the receiver to update the object's state knowing that the sender was not able to access new state or alter the old state underneath.

Building on the insights from J-Kernel, KaffeOS, and Singularity, our work develops principles for enforcing fault isolation in a safe language that enforces ownership. Similar to J-Kernel, we adopt wrapping of interfaces with proxies. We, however, generate proxies statically to avoid the run-time overhead. We rely on heap isolation similar to KaffeOS and Singularity. Our main reason for heap isolation is to be able to deallocate the domain's private heap without any semantic knowledge of objects inside. We borrow move semantics for the objects on the shared heap to provide clean fault isolation and at the same time support zero-copy communication from Singularity. We, however, extend it with the read-only borrow semantics which we need to support transparent domain recovery without giving up zero-copy. Since we implement RedLeaf in Rust, we benefit from its ownership discipline that allows us to enforce the move semantics for objects on the shared heap. Building on a body of research on linear types [86], affine types, alias types [18, 87], and region-based memory management [81], and being influenced by languages like Sing# [29], Vault [30], and Cyclone [43], Rust enforces ownership statically and without compromising usability of the language. In contrast to Singularity that heavily relies on the co-design of Sing# [29] and its communication mechanisms, we develop RedLeaf's isolation abstractions—exchangeable types, interface validation, and cross-domain call proxying—outside of the Rust language. This allows us to clearly articulate the minimal set of principles required to provide fault isolation, and develop a set of mechanisms implementing them independently from the language, that, arguably, allows adapting them to specific design tradeoffs. Finally, we make several design choices aimed at practicality of our system. We design and implement our isolation mechanisms for the most common, "migrating threads" model [31] instead of messages [39] to avoid a thread context switch on the critical cross-domain call path and allow a more natural programming idiom, e.g., in RedLeaf domain interfaces are just Rust traits.

3 RedLeaf Architecture

RedLeaf is structured as a microkernel system that relies on lightweight language-based domains for isolation (Figure 1). The microkernel implements functionality required to start threads of execution, memory management, domain loading, scheduling, and interrupt forwarding. A collection of isolated domains implement device drivers, personality of an operating system, i.e., the POSIX interface, and user applications (Section 4.5). As RedLeaf does not rely on hardware isolation primitives, all domains and the microkernel run in ring 0. Domains, however, are restricted to safe Rust (i.e., microkernel and trusted libraries are the only parts of RedLeaf that are allowed to use unsafe Rust extensions).

We enforce the heap isolation invariant between domains. To communicate, domains allocate shareable objects from a global *shared heap* and exchange special pointers, remote references (`RRef<T>`), to objects allocated on the shared heap (Section 3.1). The ownership discipline allows us to implement lightweight zero-copy communication across isolated domains (Section 3.1).

Domains communicate via normal, typed Rust function invocations. Upon cross-domain invocation, the thread moves between domains but continues execution on the same stack. Domain developers provide an interface definition for the

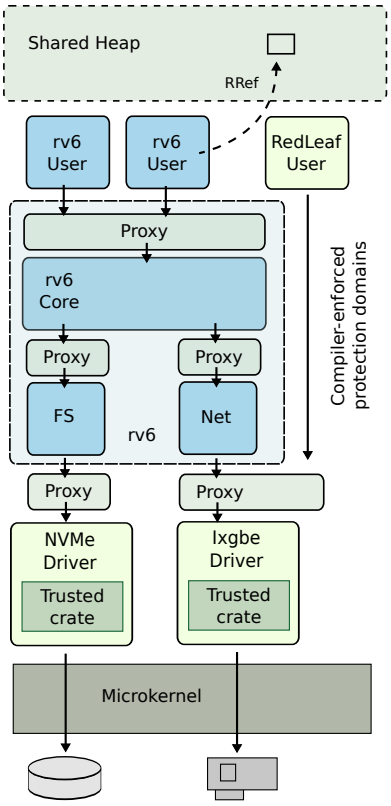


Figure 1: RedLeaf architecture

domain's entry point and its interfaces. The RedLeaf IDL compiler automatically generates code for creating and initializing domains and checks the validity of all types passed across domain boundaries (Section 3.1.5).

RedLeaf mediates all cross-domain communication with trusted proxy objects. Proxies are automatically generated from the IDL definitions by the IDL compiler (Section 3.1.5). On every domain entry, the proxy checks if a domain is alive and if so, it creates a lightweight continuation that allows us to unwind execution of the thread if the domain crashes.

In RedLeaf references to objects and traits are capabilities. In Rust, a trait declares a set of methods that a type must implement hence providing an abstraction of an interface. To expose their functionality, domains exchange references to traits via cross-domain calls. We rely on capability-based access control [76] to enforce the principle of least privilege and enable flexible operating system organizations: e.g., we implement several scenarios in which applications talk to the device driver directly bypassing the kernel, and even can link against device driver libraries leveraging DPDK-style user-level device driver access.

Protection model The core assumptions behind RedLeaf are that we trust (1) the Rust compiler to implement language safety correctly, and (2) Rust core libraries that use unsafe code, e.g., types that implement interior mutability, etc. RedLeaf's TCB includes the microkernel, a small set of trusted RedLeaf crates required to implement hardware interfaces and low-level abstractions, device crates that provide a safe interface to hardware resources, e.g., access to DMA buffers, etc., the RedLeaf IDL compiler, and the RedLeaf trusted compilation environment. At the moment, we do not address vulnerabilities in unsafe Rust extensions, but again speculate that eventually all unsafe code will be verified for functional correctness [5, 8, 82]. Specifically, the RustBelt project provides a guide for ensuring that unsafe code is encapsulated within a safe interface [44].

We trust devices to be non-malicious. This requirement can be relaxed in the future by using IOMMUs to protect physical memory. Finally, we do not protect against side-channel attacks; while these are important, addressing them is simply beyond the scope of the current work. We speculate that hardware counter-measures to alleviate the information leakage will find their way in the future CPUs [41].

3.1 Domains and Fault Isolation

In RedLeaf domains are units of information hiding, fault isolation, and composition. Device drivers, kernel subsystems, e.g., file system, network stack, etc., and user programs are loaded as domains. Each domain starts with a reference to a microkernel system-call interface as one of its arguments. This interface allows every domain to create threads of execution, allocate memory, create synchronization objects, etc. By default, the microkernel system call interface is the only authority of the domain, i.e., the only interface through which the domain can affect the rest of the system. Domains however can define a custom type for an entry function requesting additional references to objects and interfaces to be passed when it is created. By default, we do not create a new thread of execution for the domain.

Every domain, however, can create threads from the `init` function called by the microkernel when the domain is loaded. Internally, the microkernel keeps track of all resources created on behalf of each domain: allocated memory, registered interrupt threads, etc. Threads can outlive the domain creating them as they enter other domains where they can run indefinitely. Those threads continue running until they return to the crashed domain and it is the last domain in their continuation chain.

Fault isolation RedLeaf domains provide support for *fault-isolation*. We define fault isolation in the following manner. We say that a domain *crashes* and needs to be terminated when one of the threads that enters the domain panics. Panic potentially leaves objects reachable from inside the domain in an inconsistent state, making further progress of any of the threads inside the domain impractical (i.e., even if threads do not deadlock or panic, the results of the computation are undefined). Then, we say that the *fault is isolated* if the following conditions hold. First, we can unwind all threads running inside the crashing domain to the domain entry point and return an error to the caller. Second, subsequent attempts to

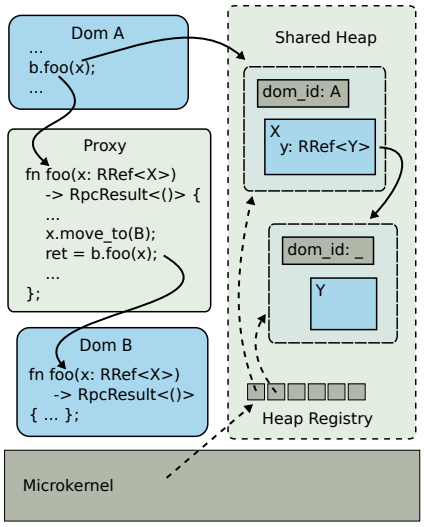


Figure 2: Inter-domain communication. Domain `A` invokes method `foo()` of domain `B`. The proxy that interposes on the invocation moves the ownership of the object pointed by `x` between domains.

invoke the domain return errors but do not violate safety guarantees or result in panics. Third, all resources of the crashed domain can be safely deallocated, i.e., other domains do not hold references into the heap of the crashed domain (heap isolation invariant), and we can reclaim all resources owned by the domain without leaks. Fourth, threads in other domains continue execution, and can continue accessing objects that were allocated by the crashed domain, but were moved to other domains before the crash.

Enforcing fault isolation is challenging. In RedLeaf isolated subsystems export complex, semantically rich interfaces, i.e., domains are free to exchange references to interfaces and hierarchies of objects. We make several design choices that allow us to cleanly encapsulate domain's state and yet support semantically rich interfaces and zero-copy communication.

3.1.1 Heap Isolation and Sharing

Private and shared heaps To provide fault isolation across domains and ensure safe termination of domains, we enforce *heap isolation* across domains, i.e., objects allocated on the private heap, stack, or global data section of the domain can not be reached from outside of the domain. This invariant allows us to safely terminate any domain at any moment of execution. Since no other domain holds pointers into the private heap of a terminated domain, it is safe to deallocate the entire heap.

To support efficient cross-domain communication, we provide a special, global *shared heap* for objects that can be sent across domains. Domains allocate objects on the shared heap in a way similar to the normal heap allocation with the Rust `Box<T>` type that allocates a value of type `T` on the heap. We construct a special type, *remote reference* or `RRef<T>`, that allocates a value of type `T` on the shared heap (Figure 2). `RRef<T>` consists of two parts: a small metadata and the value itself. The `RRef<T>` metadata contains an identifier of the domain currently owning the reference, borrow counter, and type information for the value. The `RRef<T>` metadata along with the value are allocated on the shared heap that allows `RRef<T>` to outlive the domain that originally allocates it.

Memory allocation on the domain heap To provide encapsulation of domain's private heap, we implement a two-level memory allocation scheme. At the bottom, the microkernel provides domains with an interface for allocating untyped coarse-grained memory regions (larger than one page). Each coarse-grained allocation is recorded in the *heap registry*. To serve fine-grained typed allocations on the domain's private heap, each domain links against a trusted crate that provides the Rust memory allocation interface, `Box<T>`. Domain heap allocations follow the rules of the Rust's ownership discipline, i.e., objects are deallocated when they go out of scope. The two-level scheme has the following benefit: allocating only large memory regions, the microkernel records all memory allocated by the domain without significant performance overheads. If the domain panics, the microkernel walks the registry of all untyped memory regions allocated by the allocator assigned to the domain and deallocates them without calling any destructors. Such untyped, coarse-grained deallocation is safe as we ensure the heap isolation invariant: other domains have no references into the deallocated heap.

3.1.2 Exchangeable Types

Objects allocated on the shared heap are subject to the following rule: they can be composed of only of *exchangeable* types. Exchangeable types enforce the invariant that objects on the shared heap cannot have pointers into private or shared heaps, but can have `RRef<T>`s to other objects allocated on the shared heap. RedLeaf's IDL compiler validates this invariant when generating interfaces of the domain (Section 3.1.5). We define exchangeable types as the following set: 1) `RRef<T>` itself, 2) a subset of Rust primitive `Copy` types, e.g., `u32`, `u64`, but not references in the general case, nor pointers, 3) anonymous (tuples, arrays) and named (enums, structs) composite types constructed out of exchangeable types, 4) references to traits with methods that receive exchangeable types. Also, all trait methods are required to follow the following calling convention that requires them to return the `RpcResult<T>` type to support clean abort semantics for threads returning from crashing domains (Section 3.1). The IDL checks interface definition and validates that all types are well-formed (Section 3.1.5).

3.1.3 Ownership Tracking

In RedLeaf `RRef<T>`s can be freely passed between domains. We allow `RRef<T>`s to be moved or borrowed immutably. However, we implement an ownership discipline for `RRef<T>`s that is enforced on cross-domain invocations. Ownership tracking allows us to safely deallocate objects on the shared heap owned by a crashing domain. The metadata section of the

RRef<T> keeps track of the owner domain and the number of times it was borrowed immutably on cross-domain invocations.

Initially, RRef<T> is owned by the domain that allocates the reference. If the reference is *moved* to another domain in a cross-domain call, we change the owner identifier inside RRef<T> moving ownership from one domain to another. All cross-domain communication is mediated by trusted proxies, so we can securely update the owner identifier from the proxy. Rust's ownership discipline ensures that there is always only one remote reference to the object inside the domain, hence when the reference is moved between domains on a cross-domain call, the caller loses access to the object passing it to the callee. If the reference is *borrowed immutably* in a cross-domain call, we do not change the owner identifier inside RRef<T>, but instead increment the counter that tracks the number of times RRef<T> was borrowed.

Recursive references RRef<T>s can form hierarchies of objects. To avoid moving all RRef<T>s in the hierarchy recursively on a cross-domain invocation, only the root of the object hierarchy has a valid owner identifier (in Figure 2 only object X has a valid domain identifier A, object Y does not). Upon a cross-domain call, the root RRef<T> is updated by the proxy which changes the domain identifier to move ownership of the RRef<T> between domains. This requires a special scheme for deallocating RRef<T>s in case of a crash: we scan the entire RRef<T> registry to clean up resources owned by a crashing domain. To prevent deallocation of children objects of the hierarchy, we rely on the fact that they do not have a valid RRef<T> identifier (we skip them during the scan). The drop method of the root RRef<T> object walks the entire hierarchy and deallocates all children objects (RRef<T>s cannot form cycles). Note, we should carefully handle the case when an RRef<T> is taken out of the hierarchy. To deallocate this RRef<T> correctly we need to assign it a valid domain identifier, i.e., Y gets a proper domain identifier when it is moved out from X. We mediate RRef<T> field assignments with trusted accessor methods. We generate accessor methods that provide the only way to take out an RRef<T> from an object field. This allows us to mediate the move operation and update the domain identifier for the moved RRef<T>. Note that accessors cannot be enforced for the unnamed composite types, e.g., arrays and tuples. For these types we update ownership of all composite elements upon crossing the domain boundary.

Reclaiming shared heap Ownership tracking allows us to deallocate objects that are currently owned by the crashing domain. We maintain a global registry of all allocated RRef<T>s (Figure 2). When a domain panics, we walk through the registry and deallocate all references that are owned by the crashing domain. We defer deallocation if RRef<T> was borrowed until the borrow count drops to zero. Deallocation of each RRef<T> requires that we have a drop method for each RRef<T> type and can identify the type of the reference dynamically. Each RRef<T> has a unique type identifier generated by the IDL compiler (the IDL knows all RRef<T> types in the system as it generates all cross-domain interfaces). We store the type identifier along with the RRef<T> and invoke the appropriate drop method to correctly deallocate any, possibly, hierarchical data structure on the shared heap.

3.1.4 Cross-Domain Call Proxying

To enforce fault isolation, RedLeaf relies on *invocation proxies* to interpose on all cross-domain invocations (Figure 2). A proxy object exposes an interface identical to the interface it mediates. Hence the proxy interposition is transparent to the user of the interface. To ensure isolation and safety, the proxy implements the following inside each wrapped function: 1) The proxy checks if the domain is alive before performing the invocation. If the domain is alive, the proxy records the fact that the thread moves between domains by updating its state in the microkernel. We use this information to unwind all threads that happen to execute inside the domain when it crashes. 2) For each invocation, the proxy creates a lightweight continuation that captures the state of the thread right before the cross-domain invocation. The continuation allows us to unwind execution of the thread, and return an error to the caller. 3) The proxy moves ownership of all RRef<T>s passed as arguments between domains, or updates the borrow count for all references borrowed immutably. 4) Finally, the proxy wraps all trait references passed as arguments: the proxy creates a new proxy for each trait and passes the reference to the trait implemented by that proxy.

Thread unwinding To unwind execution of a thread from a crashing domain, we capture the state of the thread right before it enters the callee domain. For each function of the trait mediated by the proxy, we utilize an assembly trampoline that saves all general registers into a *continuation*. The microkernel maintains a stack of continuations for each thread. Each continuation contains the state of all general registers and a pointer to an error handling function that has the signature identical to the function exported by the domain's interface. If we have to unwind the thread, we restore the stack to the state captured by the continuation, and invoke the error handling function on the same stack and with the same values of general registers. The error handling function returns an error to the caller.

To cleanly return an error in case of a crash, we enforce the following calling convention for all cross-domain invocations: every cross-domain function must return RpcResult<T>, an enumerated type that either holds the returned value or an error (Figure 3). This allows us to implement the following invariant: functions unwound from the crashed domain never return corrupted data, but instead return an RpcResult<T> error.

3.1.5 Interface Validation

RedLeaf's IDL compiler is responsible for validation of domain interfaces and generation of proxy code required for enforcing the ownership discipline on the shared heap. RedLeaf

```
pub trait BDev {
  fn read(&self, block: u32, data: RRef<[u8; BSIZE]>)
        -> RpcResult<RRef<[u8; BSIZE]>>;
  fn write(&self, block: u32, data: &RRef<[u8; BSIZE]>)
        -> RpcResult<()>;
}

#[create]
pub trait CreateBDev {
  fn create(&self, pci: Box<dyn PCI>)
    -> RpcResult<(Box<dyn Domain>, Box<dyn BDev>)>
}
```

Figure 3: `BDev` domain IDL interface definitions.

IDL is a subset of Rust extended with several attributes to control generation of the code (Figure 3). This design choice allows us to provide developers with the familiar Rust syntax and also re-use Rust's parsing infrastructure.

To implement an abstraction of an interface, we rely on Rust's *traits*. Traits provide a way to define a collection of methods that a type has to implement to satisfy the trait, hence defining a specific behavior. For example, the `BDev` trait requires any type that provides it to implement two methods: `read()` and `write()` (Figure 3). By exchanging references to trait objects domains connect to the rest of the system and establish communication with other domains.

Each domain provides an IDL definition for the *create* trait that allows any domain that has access to this trait to create domains of this type (Figure 3). Marked with the `#[create]` attribute, the create trait both defines the type of the domain entry function, and the trait that can be used to create the domain. Specifically, the entry function of the `BDev` domain takes the `PCI` trait as an argument and returns a pointer to the `BDev` interface. Note that when the `BDev` domain is created along with the `BDev` interface, the microkernel also returns the `Domain` trait that allows creator of the domain to control it later. The IDL generates Rust implementations of both the create trait and the microkernel code used to create the domain of this type.

Interface validation We perform interface validation as a static analysis pass of the IDL compiler. The compiler starts by parsing all dependent IDL files creating a unified abstract syntax tree (AST), which is then passed to validation and generation stages. During the interface validation pass, we use the AST to extract relevant information for each type that we validate. Essentially, we create a graph that encodes information about all types and relationships between them. We then use this graph to verify that each type is exchangeable and that all isolation constraints are satisfied: methods of cross-domain interfaces return `RpcResult<T>`, etc.

3.2 Zero-copy Communication

A combination of the Rust's ownership discipline and the single-ownership enforced on the shared heap allows us to provide isolation without sacrificing end-to-end zero-copy across the system. To utilize zero-copy communication, domains allocate objects on the shared heap with using the `RRef<T>` type. On every cross-domain invocation a mutable reference (a reference that provides writable access to the object) is moved between domains, or an immutable reference can be borrowed. If the invocation succeeds, i.e., the callee domain does not panic, a set of `RRef<T>`s might be returned by the callee moving the ownership to the caller. In contrast to Rust itself, we do not allow borrowing of mutable references. Borrowing of mutable references may result in an inconsistent state in the face of a domain crash when damaged objects are returned to the caller after the thread is unwound. Hence, we require all mutable references to be moved and returned explicitly. If a domain crashes, instead of a reference an `RpcResult<T>` error is returned.

Zero-copy is challenging in the face of crashing domains and the requirement to provide transparent recovery. A typical recovery protocol re-starts the crashing domain and re-issues the failing domain call, trying to conceal the crash from the caller. This often requires that objects passed as arguments in the re-started invocation are available inside the recovery domain. It is possible to create a copy of each object before each invocation, but this introduces significant overhead. To recover domains without additional copies, we rely on support for immutable borrowing of `RRef<T>`s on cross-domain invocations. For example, the `write()` method of the `BDev` interface borrows an immutable reference to the data written to the block device (Figure 3). If an immutable reference is borrowed by the domain, Rust's type system guarantees that the domain cannot modify the borrowed object. Hence, even if the domain crashes, it is safe to return the unmodified read-only object to the caller. The caller can re-issue the invocation as part of the recovery protocol providing the immutable reference as an argument again. This allows implementing transparent recovery without creating backup copies of arguments on each invocation that can potentially crash.

4 Implementation

While introducing a range of novel abstractions, we guide the design of RedLeaf by principles of practicality and performance. To a degree, RedLeaf is designed as a replacement for full-featured, commodity kernels like Linux.

4.1 Microkernel

The RedLeaf microkernel provides a minimal interface for creating and loading isolated domains, threads of execution, scheduling, low-level interrupt dispatch, and memory management. RedLeaf implements memory management mechanisms similar to Linux—a combination of buddy [46] and slab [16] allocators provides an interface for heap allocation inside the microkernel (the `Box<T>` mechanism). Each domain runs its own allocator internally and requests regions of memory directly from the kernel buddy allocator.

We implement the low-level interrupt entry and exit code in assembly. While Rust provides support for the *x86-interrupt* function ABI (a way to write a Rust function that takes the

x86 interrupt stack frame as an argument), in practice, it is not useful as we need the ability to interpose on the entry and exit from the interrupt, for example, to save all CPU registers.

In RedLeaf device drivers are implemented in user domains (the microkernel itself does not handle any device interrupts besides timer and NMI). Domains register threads as interrupt handlers for device-generated interrupts. For each external interrupt, the microkernel maintains a list of threads waiting for an interrupt. The threads are put back on the scheduler run queue when the interrupt is received.

4.2 Dynamic Domain Loading

In RedLeaf domains are compiled independently from the kernel and are loaded dynamically. Rust itself provides no support for dynamic extensions (except Splinter [47], existing Rust systems statically link all the code they execute [7, 50, 68]). Conceptually, the safety of dynamic extensions relies on the following invariant: types of all data structures that cross a domain boundary, including the type of the entry point function, and all types passed through any interfaces reachable through the entry function are the same, i.e., have identical meaning and implementation, across the entire system. This ensures that even though parts of the system are compiled separately type safety guarantees are preserved across domain boundaries.

To ensure that types have the same meaning across all components of the system, RedLeaf relies on a *trusted compilation environment*. This environment allows the microkernel to check that domains are compiled against the same versions of IDL interface definitions, and with the same compiler version, and flags. When a domain is compiled, the trusted environment signs the fingerprint that captures all IDL files, and a string of compiler flags. The microkernel verifies the integrity of the domain when it is loaded. Additionally, we enforce that domains are restricted to only safe Rust, and link against a white-listed set of Rust libraries.

Code generation Domain creation and loading rely on the code generated by the IDL compiler (Figure 4). IDL ensures safety at domain boundaries and allows support for user-defined domain interfaces. From the definitions of domain interfaces (Figure 4, ①) and its create function (②) the IDL generates the following code: 1) Rust implementations of all interfaces (③) and the create (④) trait, 2) a trusted entry point function (⑤) that is placed in the domain's build tree and compiled along with the rest of the domain to ensure that domain's entry function matches the domain create code, hence preserving safety on the domain boundary, 3) a microkernel domain create function that creates domains with a specific type signature of the entry point function (⑥), and 4) implementation of the proxy for this interface (⑦). By controlling the generation of the entry point, we ensure that the types of the entry function inside the microkernel and inside the domain match. If a domain tries to violate safety by changing the type of its entry function the compilation fails.

4.3 Safe Device Drivers

In RedLeaf device drivers are implemented as regular domains with no additional privileges. Like other domains they are restricted to the safe subset of Rust. To access the hardware, we provide device drivers with a collection of trusted crates that implement a safe interface to the hardware interface of the device, e.g., access to device registers and its DMA engines. For example, the ixgbe device crate provides access to the BAR region of the device, and abstracts its submit and receive queues with the collection of methods for adding and removing requests from the buffers.

Device driver domains are created by the `init` domain when the system boots. Each PCI device takes a reference to the PCI trait that is implemented inside the `pci` domain. Similar to other driver domains, the PCI driver relies on a trusted crate to enumerate all hardware devices on the bus. The trusted crate constructs `BARAddr` objects that contain addresses of PCI BAR regions. We protect each `BARAddr` object with a custom type, so it can only be used inside the trusted device crate that implements access to this specific BAR region. The `pci` domain probes device drivers with matching device identifiers. The driver receives a reference to the `BARAddr` object and starts accessing the device via its trusted crate.

4.4 Device Driver Recovery

Lightweight isolation mechanisms and clean domain interfaces allow us to implement transparent device driver recovery with shadow drivers [78]. We develop shadow drivers as normal unprivileged RedLeaf domains. Similar to proxy objects, the shadow driver wraps the interface of the device driver and exposes an identical interface. In contrast to the proxy which is relatively simple and can be generated from the IDL definition, the shadow driver is intelligent as it implements a driver-specific recovery protocol. The shadow driver interposes on all communication with the driver. During normal operation, the shadow passes all calls to the real device driver. However, it saves all information required for the recovery of the driver (e.g., references to PCI trait, and other parts of the device initialization protocol). If the driver crashes, the shadow driver receives an error from the proxy domain. The proxy itself receives the error when the thread is unwound through the continuation mechanism. Instead of returning an error to its caller, the shadow triggers the domain recovery protocol. It creates a new driver domain and replays its initialization protocol, by interposing on all external communication of the driver.

4.5 Rv6 Operating System Personality

To evaluate the generality of RedLeaf's abstractions, we implemented Rv6, a POSIX-subset operating system on top of RedLeaf. At a high-level, Rv6 follows the implementation of the xv6 operating system [73], but is implemented as a collection of isolated RedLeaf domains. Specifically, we implement Rv6 as the following domains: the core kernel, file system,

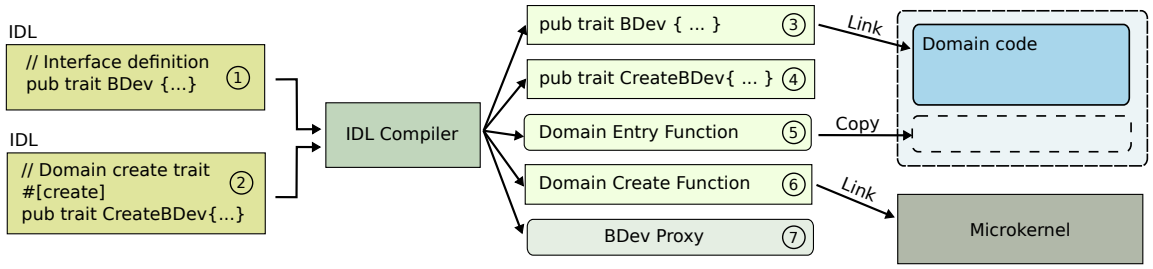


Figure 4: IDL code generation

network stack subsystem, network and disk device drivers, and collection of user domains. User domains communicate with the core kernel through the Rv6 system call interface. The core kernel dispatches the system call to either the file system or a network stack. The file system itself communicates with one of the RedLeaf block device drivers to get access to disk. We implemented three block device drivers: in-memory, AHCI, and NVMe. The file system implements journaling, buffer cache, inode, and naming layers. The network subsystem implements the TCP/IP stack and connects to the network device driver (we currently implement only one driver that supports a 10Gbps Intel Ixgbe device). We do not support the full semantics of the `fork()` system call as we do not rely on address spaces and hence cannot virtualize and clone the address space of the domain. Instead, we provide a combination of `create` system calls that allow user applications to load and start new domains [10]. Rv6 boots into a shell that supports pipes and I/O redirection and can start other applications similar to a typical UNIX system.

5 Evaluation

We conduct all experiments in the openly-available CloudLab network testbed [72].[2] For network-based experiments, we utilize two CloudLab c220g2 servers configured with two Intel E5-2660 v3 10-core Haswell CPUs running at 2.6 GHz, 160 GB RAM, and a dual-port Intel X520 10Gb NIC. We run our NVMe benchmarks on a CloudLab d430 node that is configured with two 2.4 GHz 64-bit 8-Core E5-2630 Haswell CPUs, and a PCIe-attached 400GB Intel P3700 Series SSD. Linux machines run 64-bit Ubuntu 18.04 with a 4.8.4 kernel configured without any speculative execution attack mitigations as recent Intel CPUs address a range of speculative execution attacks in hardware. All RedLeaf experiments are performed on bare-metal hardware. In all the experiments, we disable hyper-threading, turbo boost, CPU idle states, and frequency scaling to reduce the variance in benchmarking.

5.1 Overheads of Domain Isolation

Language based isolation versus hardware mechanisms To understand the benefits of language-based isolation over traditional hardware mechanisms, we compare RedLeaf's cross-domain calls with the synchronous IPC mechanism implemented by the seL4 microkernel [27], and a recent kernel-isolation framework that utilizes VMFUNC-based extended page table (EPT) switching [62]. We choose seL4 as it implements the fastest synchronous IPC across several modern microkernels [58]. We configure seL4 without meltdown mitigations. On the c220g2, server seL4 achieves the cross-domain invocation latency of 834 cycles (Table 1).

Operation	Cycles
seL4	834
VMFUNC	169
VMFUNC-based call/reply invocation	396
RedLeaf cross-domain invocation	124
RedLeaf cross-domain invocation (passing an `RRef<T>`)	141
RedLeaf cross-domain invocation via shadow	279
RedLeaf cross-domain via shadow (passing an `RRef<T>`)	297

Table 1: Language-based cross-domain invocation vs hardware isolation mechanisms.

Recent Intel CPU introduces two new hardware isolation primitives—memory protection keys (MPK) and EPT switching with VM functions—provide support for memory isolation with overheads comparable to system calls [83] (99-105 cycles for MPK [38, 83] and 268-396 cycles for VMFUNC [38, 58, 62, 83]). Unfortunately, both primitives require complex mechanisms to enforce isolation, e.g., binary rewriting [58, 83], protection with hardware breakpoints [38], execution under control of a hypervisor [54, 58, 62]. Moreover, since neither MPK nor EPT switching are designed to support isolation of privileged ring 0 code, additional techniques are required to ensure isolation of kernel subsystems [62].

To compare the performance of EPT-based isolation with language-based techniques we use in RedLeaf, we configure LVDs, a recent EPT-based kernel isolation framework [62] to perform ten million cross-domain invocations and measure the latency in cycles with the `RDTSC` instruction. In LVDs, the cross-domain call relies on the VMFUNC instruction to switch the root of the EPT and selects a new stack in the callee domain. LVDs, however, require no additional switches of a privilege level or a page-table. A single VMFUNC instruction takes 169 cycles, while a complete call/reply invocation takes 396 cycles on the c220g2 server (Table 1).

In RedLeaf, a cross-domain call is initiated by invoking

[2] RedLeaf is available at `https://mars-research.github.io/redleaf`.

the trait object provided by the proxy domain. The proxy domain uses a microkernel system call to move the thread from the callee to the caller domain, creates continuation to unwind the thread to the entry point in case the invocation fails, and invokes the trait of the callee domain. On the return path, a similar sequence moves the thread from the callee domain back into the caller. In RedLeaf, a null cross-domain invocation via a proxy object (Table 1) introduces an overhead of 124 cycles. Saving the state of the thread, i.e., creating continuation, takes 86 cycles as it requires saving all general registers. Passing one `RRef<T>` adds an overhead of 17 cycles as `RRef<T>` is moved between domains. To understand the low-level overhead of transparent recovery, we measure the latency of performing the same invocation via a shadow domain. In case of a shadow the invocation crosses two proxies and a user-built shadow domain and takes 286 cycles due to additional crossing of proxy and shadow domains.

Most recent Intel CPUs implement support for ring 0 enforcement of memory protection keys, protection keys supervisor (PKS) [3], finally enabling low-overhead isolation mechanism for the privileged kernel code. Nevertheless, even with low-overhead hardware isolation mechanisms, a zero-copy fault-isolation scheme requires ownership discipline for shared objects that arguably requires support from the programming language, i.e., either a static analysis [39] or a type system that can enforce single-ownership.

Overheads of Rust Memory safety guarantees of Rust come at a cost. In addition to the checks required to ensure safety at runtime, some Rust abstractions have a non-zero runtime cost, e.g., types that implement interior mutability, option types, etc. To measure the overheads introduced by Rust language itself, we develop a simple hash table that uses an open-addressing scheme and relies on the Fowler–Noll–Vo (FNV) hashing function with linear probing to store eight byte keys and values. Using the same hashing logic, we develop three implementations: 1) in plain C, 2) in idiomatic Rust (the style encouraged by the Rust programming manual), and 3) in C-style Rust that essentially uses C programming idioms but in Rust. Specifically, in C-style Rust, we avoid 1) using higher-order functions and 2) the `Option<T>` type that we utilize in the idiomatic code to distinguish between the occupied and unoccupied entries in the table. Without the `Option<T>` type that adds at least one additional byte to the key-value pair, we benefit from a tight, cache-aligned representation of key-value pairs in memory to avoid additional cache misses. We vary the number of entries in the hash table from 2^{12} to 2^{26} and keep the hash-table 75% full. On most hash table sizes, our implementation in idiomatic Rust remains 25% slower than the one in plain C, whereas C-style Rust performs equal to or even better than plain C, although by only 3-10 cycles (Figure 5). We attribute this to a more compact code generated by the Rust compiler (47 instructions on the critical get/set path in C-style Rust versus 50 instructions in C).

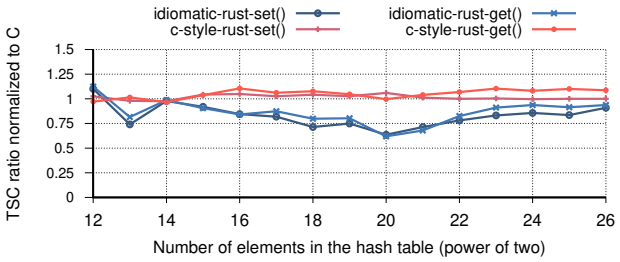


Figure 5: C vs Rust performance comparison

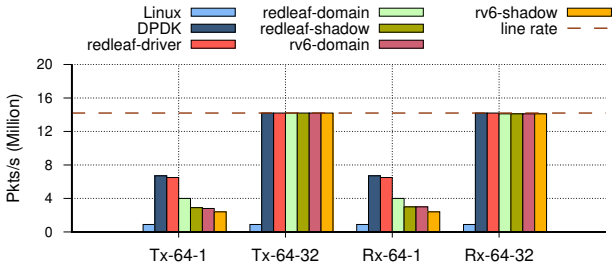


Figure 6: Ixgbe driver performance

5.2 Device Drivers

One of the critical assumptions behind RedLeaf is that Rust's safety is practical for development of the fastest subsystems of a modern operating system kernel. Today, operating with latencies of low hundreds of cycles per I/O request, device drivers that provide access to high-throughput I/O interfaces, network adapters and low-latency non-volatile PCIe-attached storage, have the tightest performance budgets among all kernel components. To understand if overheads of Rust's zero-cost abstractions allow the development of such low-overhead subsystems, we develop two device drivers: 1) an Intel 82599 10Gbps Ethernet driver (Ixgbe), and 2) an NVMe driver for PCIe-attached SSDs.

5.2.1 Ixgbe Network Driver

We compare the performance of RedLeaf's Ixgbe driver with the performance of a highly-optimized driver from the DPDK user-space packet processing framework [21] on Linux. Both DPDK and our driver work in polling mode, allowing them to achieve peak performance. We configure RedLeaf to run several configurations: 1) `redleaf-driver`: the benchmark application links statically with the driver (this configuration is closest to user-level packet frameworks like DPDK; similarly, we pass-through the Ixgbe interface directly to the RedLeaf); 2) `redleaf-domain`: the benchmark application runs in a separate domain, but accesses the driver domain directly via a proxy (this configuration represents the case when the network device driver is shared across multiple isolated applications [38]); 3) `rv6-domain`: the benchmark application runs as an Rv6 program, it first enters the Rv6 with a system call and then calls into the driver (this configuration is analogous to a setup of a commodity operating system kernel in which user

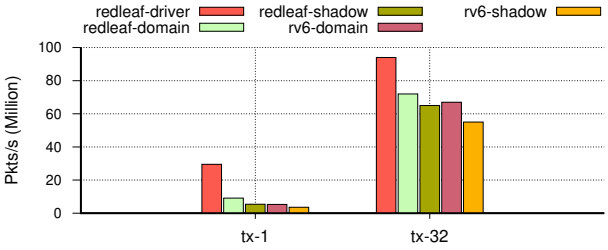


Figure 7: Software-only `nullnet` driver performance

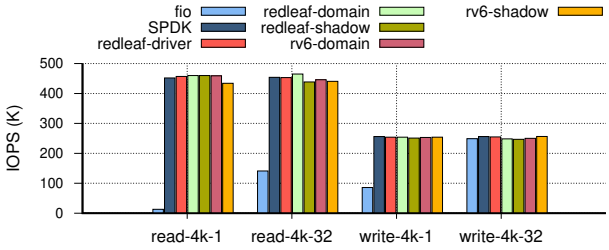


Figure 8: Performance of the NVMe driver

applications access I/O interfaces via a kernel network stack). Further, we run the last two configurations with and without the shadow driver (`redleaf-shadow` and `rv6-shadow`), which introduces an additional domain crossing into the shadow (these two configurations evaluate overheads of the transparent driver recovery). In all our tests, we pin the application thread to a single CPU core.

We send 64 byte packets and measure the performance on two batch sizes: 1 and 32 packets (Figure 6). For packet receive tests, we use a fast packet generator from the DPDK framework to generate packets at line-rate. On packet transmit and receive tests, Linux achieves 0.89 Mpps due to its overly general network stack and synchronous socket interface (Figure 6). On a batch of one, DPDK achieves 6.7 Mpps and is 7% faster than RedLeaf (6.5 Mpps) for both RX and TX paths (Figure 6). On a batch of 32 packets, both drivers achieve the line-rate performance of a 10GbE interface (14.2 Mpps). To understand the impact of cross-domain invocations, we run the benchmark application as a separate domain (`redleaf-domain`) and as an Rv6 program (`rv6-domain`). The overhead of domain crossings is apparent on a batch size of one, where RedLeaf can send and receive packets at the rate of 4 Mpps per-core with one domain crossing (`redleaf-domain`) and 2.9 Mpps if the invocation involves shadow domain (`redleaf-shadow`). With two domain crossings, the performance drops to 2.8 Mpps (`rv6-domain`) and 2.4 Mpps if the driver is accessed via a shadow (`rv6-shadow`). On a batch of 32 packets, the overhead of domain crossings disappears as all configurations saturate the device.

Nullnet To further investigate the overheads of isolation without the limits introduced by the device itself, we develop a software-only `nullnet` driver that simply returns the packet to the caller instead of queuing it to the device (Figure 7). On a batch of one, the overheads of multiple domain crossings limit the theoretical performance of `nullnet` driver from 29.5 Mpps per-core that can be achieved if the application is linked statically with the driver (`redleaf-driver`) to 5.3 Mpps when `nullnet` is accessed from the Rv6 application (`rv6-domain`). Adding a shadow driver lowers this number to 3.6 Mpps (`rv6-shadow`). Similarly, on a batch of 32 packets, `nullnet` achieves 94 Mpps if the applicaiton is run in the same domain as the driver. The performance drops to 67 Mpps when the benchmark code runs as an Rv6 application (`rv6-domain`), and to 55 Mpps if the Rv6 application involves a shadow driver (`rv6-shadow`).

5.2.2 NVMe Driver

To understand the performance of RedLeaf's NVMe driver, we compare it with the multi-queue block driver in the Linux kernel and a well-optimized NVMe driver from the SPDK storage framework [42]. Both SPDK and RedLeaf drivers work in polling mode. Similar to Ixgbe, we evaluate several configurations: 1) statically linked (`redleaf-driver`); 2) requiring one domain crossing (`redleaf-domain`); and 3) running as an Rv6 user program (`rv6-domain`). We run the last two configurations with and without the shadow driver (`redleaf-shadow` and `rv6-shadow`). All tests are limited to a single CPU core.

We perform sequential read and write tests with a block size of 4KB on a batch size of 1 and 32 requests (Figure 8). On Linux, we use `fio`, a fast I/O generator; on SPDK and RedLeaf, we develop similar benchmark applications that submit a set of requests at once, and then poll for completed requests. To set an optimal baseline for our evaluation, we chose the configuration parameters that can give us the fastest path to the device. Specifically, on Linux, we configure `fio` to use the asynchronous `libaio` library to overlap I/O submissions, and bypass the page cache with the `direct` I/O flag.

On sequential read tests, `fio` on Linux achieves 13K IOPS and 141K IOPS per-core on the batch size of 1 and 32 respectively (Figure 8). On a batch size of one, the RedLeaf driver is 1% faster (457K IOPS per-core) than SPDK (452K IOPS per-core). Both drivers achieve maximum device read performance. SDPK is slower as it performs additional processing aimed at collecting performance statistics on each request. On a batch size of 32, the RedLeaf driver is less than 1% slower (453K IOPS versus 454K IOPS SPDK). On sequential write tests with a batch size of 32, Linux is within 3% of the device's maximum throughput of around 256K IOPS. RedLeaf is less than one percent slower (255K IOPS). Since NVMe is a slower device compared to Ixgbe, the overheads of domain crossings are minimal for both batch sizes. With one domain crossing, the performance even goes up by 0.7% (we attribute this to a varying pattern of accessing the doorbell register of the device that gets thrashed between the device and CPU).

5.3 Application Benchmarks

To understand the performance overheads of safety and isolation on application workloads, we develop several applications that traditionally depend on a fast data plane of the op-

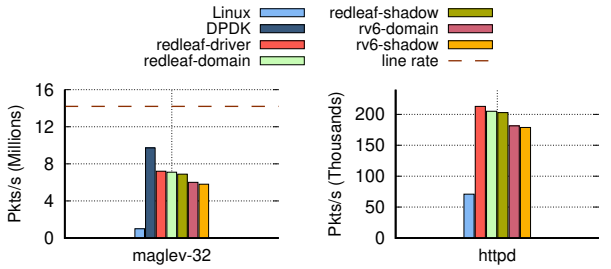


Figure 9: Performance of Maglev and Httpd

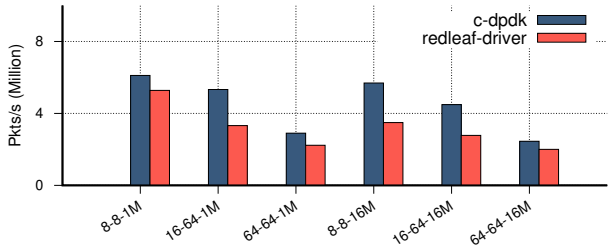


Figure 10: Key-value store

erating system kernel: 1) Maglev load balancer (`maglev`) [26], 2) a network-attached key-value store (`kv-store`), and 3) a minimal web server (`httpd`).

Maglev load-balancer Maglev is a load balancer developed by Google to evenly distribute incoming client flows among a set of backend servers [26]. For each new flow, Maglev selects one of the available backends by performing a lookup in a hash table, size of which is proportional to the number of backend servers (65,537 in our experiments). Consistent hashing allows even distribution of flows across all servers. Maglev then records the chosen backend in a hash table, a *flow tracking table*, that is used to redirect packets from the same flow to the same backend server. The size of the flow tracking table is proportional to the number of flows (we choose 1 M flows for our experiments). Processing a packet requires a lookup in the flow tracking table if it is an existing flow, or a lookup of a backend server and an insertion into the flow tracking table to record the new flow. To compare RedLeaf performance with both a commodity and the fastest possible setup, we develop C and Rust versions of the core Maglev logic. Moreover, we evaluate two C versions: one to run as a normal Linux program that uses the socket interface and another developed to work as a network function for the DPDK network processing framework [21]. In all versions we follow the same code logic and, if possible, apply the same optimizations. Again, on all setups, we restrict execution to one CPU core. Running as a Linux program, `maglev` is limited to 1 Mpps per-core due to the synchronous socket interface of the Linux kernel and a generic network stack (Figure 9). Operating on a batch of 32 packets, the `maglev` DPDK function is capable of achieving 9.7 Mpps per-core due to a well-optimized network device driver. Linked statically against the driver, RedLeaf application (`redleaf-driver`) achieves 7.2 Mpps per-core. Performance drops with additional domain crossings. Running as an Rv6 application, `maglev` can forward at 5.3 Mpps per-core without and 5.1 Mpps with the shadow domain.

Key-value store Key-value stores are de facto standard building blocks for a range of datacenter systems ranging from social networks [64] to key-value databases [23]. To evaluate RedLeaf's ability to support the development of efficient datacenter applications, we develop a prototype of a network-attached key-value store, `kv-store`. Our prototype is designed to utilize a range of modern optimizations similar to Mica [52], e.g., a user-level device driver like DPDK, partitioned design aimed at avoiding cross-core cache-coherence traffic, packet flow steering to guarantee that request is directed to the specific CPU core where the key is stored, no locks and no allocations on the request processing path, etc. Our implementation relies on a hash table that uses open addressing scheme with linear probing and the FNV hash function. In our experiments, we compare the performance of two implementations: a C version developed for DPDK, and a Rust version that executes in the same domain with the driver (`redleaf-driver`), i.e., the configuration that is closest to DPDK. We evaluate two hash table sizes: 1 M and 16 M entries with three sets of key and value pairs (`<8B,8B>`, `<16B,64B>`, `<64B,64B>`). The RedLeaf version is implemented in a C-style Rust code, i.e., we avoid Rust abstractions that have run-time overhead (e.g., `Option<T>`, and `RefCell<T>` types). This ensures that we can control the memory layout of the key-value pair to avoid additional cache misses. Despite our optimizations, RedLeaf achieves only 61-86% performance of the C DPDK version. The main reason for the performance degradation is that being implemented in safe Rust, our code uses vectors, `Vec<T>`, to represent packet data. To create a response, we need to extend this vector thrice by calling the `extend_from_slice()` function to copy the response header, key, and value into the response packet. This function checks if the vector needs to be grown and performs a copy. In contrast, the C implementation benefits from a much lighter unsafe invocation of `memcpy()`. As an exercise, we implemented the packet serialization logic with unsafe Rust typecast that allowed us to achieve 85-94% of the C's performance. However, we do not allow unsafe Rust inside RedLeaf domains.

Web server The latency of web page loading plays a critical role in both the user experience, and the rank of the page assigned by a search engine [15, 66]. We develop a prototype of a web server, `httpd`, that can serve static HTTP content. Our prototype uses a simple run-to-completion execution model that polls incoming requests from all open connections in a round-robin fashion. For each request, it performs request parsing and replies with the requested static web page. We compare our implementation with one of the de facto industry standard web servers, Nginx [63]. In our tests, we use the `wrk` HTTP load generator [1], which we configure to run with one thread and 20 open connections. On Linux, Nginx can serve

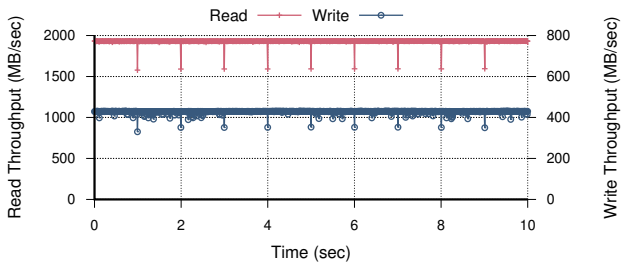


Figure 11: Block device recovery (FS write).

70.9 K requests per second, whereas our implementation of `httpd` achieves 212 K requests per second in a configuration where the application is run in the same domain as the driver (`redleaf-driver`) and network stack (Figure 9). Specifically, we benefit from low-latency access to the network stack and the network device driver. Running as an Rv6 domain, `httpd` achieves the rate of 181.4 K packets per second (178.9 K if it uses a shadow).

5.4 Device Driver Recovery

To evaluate the overheads introduced by the transparent device driver recovery, we develop a test in which an Rv6 program accesses the Rv6 file system backed by an in-memory block device. Running as an Rv6 program, the benchmark application continuously reads and writes files in the Rv6 file system using 4K blocks. The Rv6 file system accesses the block device via a shadow driver that can perform recovery of the block device in case of a crash. During the test, we trigger a crash of the block device driver every second (Figure 11). Automatic recovery triggers a small drop in performance. For reads, the throughput with and without restarts averages at 2062 MB/s and 2164 MB/s respectively (a 5% drop in performance). For writes, the total throughput averages at 356 MB/s with restarts and 423 MB/s without restarts (a 16% drop in performance).

6 Related Work

Several recent projects use Rust for building low-level high-performance systems, including data storage [33, 47, 60], network function virtualization [68], web engine [74], and several operating systems [17, 24, 50, 51], unikernels [49] and hypervisors [4, 36, 40]. Firecracker [4], Intel Cloud Hypervisor [40], and Google Chrome OS Virtual Machine Monitor [36] replace Qemu hardware emulator with a Rust-based implementation. Redox [24] utilizes Rust for development of a microkernel-based operating system (both microkernel and user-level device drivers are implemented in Rust, but are free to use unsafe Rust). The device drivers run in ring 3 and use traditional hardware mechanisms for isolation and system calls for communication with the microkernel. By and large, all these systems leverage Rust as a safe alternative to C, but do not explore the capabilities of Rust that go beyond type and memory safety.

Tock develops many principles of minimizing the use of unsafe Rust in a hardware-facing kernel code [50]. Tock is structured as a minimal core kernel and a collection of device drivers (capsules). Tock relies on Rust's language safety for isolation of the capsules (in Tock user applications are isolated with commodity hardware mechanisms). To ensure isolation, Tock forbids unsafe extensions in capsules but does not restrict sharing of pointers between capsules and the main kernel (this is similar to language systems using pointers as capabilities, e.g., SPIN [13]). As a result, a fault in any of the capsules halts the entire system. Our work builds on many design principles aimed at minimizing the amount of unsafe Rust code developed by Tock but extends them with support for fault isolation and dynamic loading of extensions. Similar to Tock, Netbricks [68] and Splinter [47] rely on Rust for isolation of network functions and user-defined database extensions. None of the systems provides support for deallocating resources of crashing subsystems, recovery, or generic exchange of interfaces and object references.

7 Conclusions

"A Journey, not a Destination" [39], Singularity OS laid the foundation for many concepts that influenced the design of Rust. In turn, by enabling the principles of fault isolation in Rust itself, our work completes the cycle of this journey. RedLeaf, however, is just a step forward, not a final design—while guided by principles of practicality and performance, our work is, first, a collection of mechanisms and an experimentation platform for enabling future system architectures that leverage language safety. Rust provides systems developers the mechanisms we were waiting for decades: practical, zero-cost safety, and a type system that enforces ownership. Arguably, the isolation that we implement is the most critical mechanism as it provides a foundation for enforcing a range of abstractions in systems with faulty and mistrusting components. By articulating principles of isolation, our work unlocks future exploration of abstractions enabled by the isolation and safety: secure dynamic extensions, fine-grained access control, least privilege, collocation of computation and data, transparent recovery, and many more.

Acknowledgments

We would like to thank USENIX ATC 2020 and OSDI 2020 reviewers and our shepherd, Michael Swift, for numerous insights helping us to improve this work. Also, we would like to thank the Utah CloudLab team, and especially Mike Hibler, for his continuous support in accommodating our hardware requests. We thank Abhiram Balasubramanian for helping with RedLeaf device drivers and Nivedha Krishnakumar for assisting us with low-level performance analysis. This research is supported in part by the National Science Foundation under Grant Numbers 1837051 and 1840197, Intel and VMWare.

References

[1] wrk - a HTTP benchmarking tool. https://github.com/wg/wrk.

[2] Erlang on Xen. http://erlangonxen.org/, 2012.

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2020. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 419–434, 2020.

[5] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust Types for Modular Specification and Verification. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, volume 3, pages 147:1–147:30.

[6] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[7] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*, pages 156–161, 2017.

[8] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. Verifying Rust Programs with SMACK. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 11138 of *Lecture Notes in Computer Science*, pages 528–535. Springer, 2018.

[9] Fred Barnes, Christian Jacobsen, and Brian Vinter. RMoX: A Raw-Metal occam Experiment. In *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 182–196, September 2003.

[10] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A Fork() in the Road. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*, page 14–22, 2019.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 49–65, October 2014.

[12] D. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp., March 1976.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, page 267–283, 1995.

[14] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The Development of the Emerald Programming Language. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*, page 11–1–11–51, 2007.

[15] Google Webmaster Central Blog. Using site speed in web search ranking. https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html.

[16] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC'94)*, page 6, 1994.

[17] Kevin Boos and Lin Zhong. Theseus: A State Spill-Free Operating System. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17)*, page 29–35, 2017.

[18] John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[19] Hank Bromley and Richard Lamson. *LISP Lore: A Guide to Programming the Lisp Machine*. Springer Science & Business Media, 2012.

[20] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys '11)*, pages 5:1–5:5, 2011.

[21] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/.

[22] Cody Cutler, M Frans Kaashoek, and Robert T Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proceedings of the 13th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI '18), pages 89–105, 2018.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, page 205–220, 2007.

[24] Redox Project Developers. Redox - Your Next(Gen) OS. http://www.redox-os.org/.

[25] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[26] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, pages 523–535, March 2016.

[27] Kevin Elphinstone and Gernot Heiser. From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, page 133–150, 2013.

[28] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[29] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-Based Communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, page 177–190, 2006.

[30] Manuel Fahndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 13–24, 2002.

[31] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC '94)*, pages 97–114, 1994.

[32] Lester J Fraim. Scomp: A Solution to the Multilevel Security Problem. *Computer*, 16(07):26–34, July 1983.

[33] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: Dynamic, Partially-Stateful Data-Flow for High-Performance Web Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, page 213–231, 2018.

[34] Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementation. *Addison-Wesley Longman Publishing Co., Inc.*, 1983.

[35] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATC '02)*, page 45–58, 2002.

[36] Google. Google Chrome OS Virtual Machine Monitor. https://chromium.googlesource.com/chromiumos/platform/crosvm.

[37] Haskell Lightweight Virtual Machine (HaLVM). http://corp.galois.com/halvm.

[38] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 489–504, July 2019.

[39] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[40] Intel. Cloud Hypervisor VMM. https://github.com/cloud-hypervisor/cloud-hypervisor.

[41] Intel. Side Channel Mitigation by Product CPU Model. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html.

[42] Intel Corporation. Storage Performance Development Kit (SPDK). https://spdk.io.

[43] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang.

Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference (ATC '02)*, pages 275–288, June 2002.

[44] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the ACM on Programming Languages (POPL)*, volume 2, pages 1–34, 2017.

[45] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019.

[46] Kenneth C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–624, October 1965.

[47] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 627–643, October 2018.

[48] Butler W. Lampson and Robert F. Sproull. An Open Operating System for a Single-User Machine. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP '79)*, page 98–105. 1979.

[49] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS'19)*, page 8–15, 2019.

[50] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, page 234–251, 2017.

[51] Alex Light. Reenix: Implementing a Unix-like operating system in Rust. Undergraduate Honors Theses, Brown University, 2015.

[52] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, pages 429–444, April 2014.

[53] John Lions. *Lions' commentary on UNIX 6th edition with source code*. Peer-to-Peer Communications, Inc., 1996.

[54] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, page 1607–1619, 2015.

[55] Peter W Madany, Susan Keohan, Douglas Kramer, and Tom Saulpaugh. JavaOS: A Standalone Java Environment. *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[56] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, March 2013.

[57] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, page 115–128, 2011.

[58] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys '19)*, 2019.

[59] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[60] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, Iterative Data Processing with Timely Dataflow. *Communications of the ACM*, 59(10):75–83, September 2016.

[61] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 269–284, July 2019.

[62] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*, page 157–171, 2020.

[63] Nginx. Nginx: High Performance Load Balancer, Web Server, and Reverse Proxy. https://www.nginx.com/.

[64] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, April 2013.

[65] Oreboot developers. Oreboot. https://github.com/oreboot/oreboot.

[66] Addy Osmani and Ilya Grigorik. Speed is now a landing page factor for Google Search and Ads. https://developers.google.com/web/updates/2018/07/search-ads-speed.

[67] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, page 305–318, 2011.

[68] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 203–216, November 2016.

[69] Matthew Parkinson. Digital Security by Design: Security and Legacy at Microsoft. https://vimeo.com/376180843, 2019. ISCF Digital Security by Design: Collaboration Development Workshop.

[70] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 1–16, October 2014.

[71] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, 1980.

[72] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[73] Robert Morris Russ Cox, Frans Kaashoek. Xv6, a simple Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2019/xv6.html, 2019.

[74] Servo, the Parallel Browser Engine Project. http://www.servo.org.

[75] Christopher Small and Margo I. Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR 30-94, Harvard University, Division of Engineering and Applied Sciences, 1994.

[76] Marc Stiegler. The E Language in a Walnut, 2000. http://www.skyhunter.com/marcs/ewalnut.html.

[77] Jeff Vander Stoep. Android: protecting the kernel. *Linux Security Summit*, 2016.

[78] Michael M Swift, Muthukaruppan Annamalai, Brian N Bershad, and Henry M Levy. Recovering Device Drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360, 2006.

[79] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, 2002.

[80] Daniel C Swinehart, Polle T Zellweger, Richard J Beach, and Robert B Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):419–490, 1986.

[81] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[82] J. Toman, S. Pernsteiner, and E. Torlak. Crust: A Bounded Verifier for Rust (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 75–80, November 2015.

[83] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pages 1221–1238, August 2019.

[84] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 469–480, June 2014.

[85] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System

for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[86] Philip Wadler. Linear Types Can Change the World! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359, 1990.

[87] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures (Extended Version). Technical Report TR2000-1787, Cornell University, March 2000.

[88] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Memory Management*, pages 1–42, 1992.

[89] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, page 31–44, 2005.

[90] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.



Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel

Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang
University of Washington

Abstract

This paper describes our experience applying formal methods to a critical component in the Linux kernel, the just-in-time compilers ("JITs") for the Berkeley Packet Filter (BPF) virtual machine. We verify these JITs using Jitterbug, the first framework to provide a precise specification of JIT correctness that is capable of ruling out real-world bugs, and an automated proof strategy that scales to practical implementations. Using Jitterbug, we have designed, implemented, and verified a new BPF JIT for 32-bit RISC-V, found and fixed 16 previously unknown bugs in five other deployed JITs, and developed new JIT optimizations; all of these changes have been upstreamed to the Linux kernel. The results show that it is possible to build a verified component within a large, unverified system with careful design of specification and proof strategy.

1 Introduction

Downloading application code into the OS kernel is a general approach to extensibility [26]. To extend the kernel, the application submits a program written in a dedicated language, and the kernel executes this program using an interpreter, or translates it into machine code for native execution via a *just-in-time (JIT) compiler* [3]. Berkeley Packet Filter (BPF) [31] is one such language, and it is used to implement a wide variety of extensions for the Linux kernel, including networking [38], security [79], and tracing [35], among many other services [18, 57].

Given the prevalence of BPF code and its execution in the OS kernel, the correctness of BPF JIT compilers (or simply "JITs") is critical for the system. Compared to the BPF interpreter, using the JITs is both more efficient and more resistant to speculative attacks [84], leading major Linux distributions to remove the BPF interpreter from the kernel in favor of the JITs [9]. But the JITs are more susceptible to subtle correctness bugs due to their complexity (§3).

This paper presents a formal approach to building JITs in the kernel with high assurance of correctness. We develop Jitterbug, a framework for writing JITs and proving them correct. Using Jitterbug, we design, implement, and verify a BPF JIT for RV32, the 32-bit RISC-V architecture [96]. We also port the existing JITs for Arm32, Arm64, RV64, x86-32, and x86-64 to Jitterbug, uncovering 16 previously unknown bugs. We write patches that fix these bugs and introduce new optimizations, all of which are verified to be correct. The BPF JIT for RV32, bug fixes, and optimizations have been upstreamed to the Linux kernel.

Jitterbug is designed to meet three competing requirements: *deployability* of verified JITs with minimal changes to the Linux kernel; *proof automation* to support rapid verification of JITs; and *separability* of verified JITs from any verification artifacts, making the resulting code auditable by kernel developers with no background in formal methods. Each of these requirements comes with its own challenges and trade-offs.

First, BPF JITs and their generated code interact with a monolithic kernel via an existing interface, which was not designed for verification. As Jitterbug emphasizes deployability, it cannot adopt the clean-slate design favored by previous verification efforts [33, 65, 81, 94] or change this interface to simplify verification. Therefore, it needs a correctness specification that is both capable of ruling out real-world bugs and amenable to verification. Developing such a specification is challenging even for clean-slate designs with strong simplifying assumptions, and it is the core technical challenge addressed by Jitterbug.

Second, verification needs to catch up with increasing functionality and optimization of BPF JITs. Jitterbug thus prioritizes proof automation to free developers from the burden of writing manual proofs and to enable rapid verification in the code review process. Prior work has shown success in scaling automated verification to systems whose code does not change in response to input [68, 70]. But verifying a JIT is particularly challenging, because it requires reasoning about not only the behavior of the JIT itself, but also that of the machine code generated by the JIT for input BPF programs.

Third, kernel development emphasizes the efficiency and clarity of source code, whereas formal development emphasizes managing code complexity to make verification tractable.

Jitterbug must resolve the tension and make the two development processes cleanly separable. While formal development can use specific tools and artifacts such as specifications, the final implementation of a JIT needs to be C code that can be reviewed assuming no knowledge of formal methods, and can be compiled using a standard toolchain.

To address these challenges, Jitterbug makes the following contributions:

- A precise stepwise specification for JIT correctness (§4). The specification models both BPF and target architectures as abstract machines, and it formulates JIT correctness as the behavioral equivalence of running the machines with a source BPF instruction and the target instructions produced by the JIT, respectively. The specification assumes that a JIT translates a single source instruction at a time. This assumption matches real-world BPF JIT implementations and obviates the need to reason about translating entire programs.
- An automated proof strategy that scales to practical BPF JITs (§5). Building on Serval [68], Jitterbug uses symbolic evaluation [10, 89] to produce a satisfiability query that encodes the semantics of a JIT implementation, the semantics of source BPF code, and the semantics of target machine code produced by the JIT. It then discharges the query using an SMT solver [21]. Since Serval was designed to reason about systems whose code is statically known, it cannot be used to verify *symbolic* instructions (e.g., with symbolic fields, at symbolic addresses) generated by the symbolic evaluation of a JIT. Jitterbug addresses this challenge with a symbolic evaluation strategy that can reason about such symbolic code.
- An approach to writing JITs in a domain-specific language (DSL) based on C (§6). The Jitterbug DSL is a *shallow embedding* of a structured subset of C in Rosette [88, 89], which extends Racket [29] for symbolic reasoning. That is, the Jitterbug DSL implements a subset of C as a Rosette library. We write new JITs in the DSL, which simplifies verification and enables synthesis of JIT optimizations [59, 82]. Jitterbug automates the step of translating JITs written in the DSL to C through an (unverified) extraction mechanism. We verify existing JITs by manually translating their C code to Rosette.
- Experience with using Jitterbug to build a BPF JIT for RV32, find and fix bugs in five existing BPF JITs, perform code review, develop optimizations, and port a JIT for a stack machine [65], all with low verification overhead (§7). One of the bugs has led to a clarification in the RISC-V instruction-set manual. We report on the iterative process of improving Jitterbug and upstreaming JIT code to the Linux kernel.

To our knowledge, Jitterbug is the first to provide a specification that rules out bugs in practical JIT implementations, and a proof strategy that scales automated verification to a class of compilers. It demonstrates the feasibility of building a verified component (i.e., the BPF JIT) within a large, unverified system under active development (i.e., the Linux kernel), through careful design of specification and proof strategy. This paper describes our design decisions and the rationale behind them (§8).

2 Related work

Code downloading for extensible systems. The Xerox Alto allows applications to customize and optimize the system through *microcode* [51, 85]. It pioneered the use of *packet filters* for demultiplexing, debugging, and monitoring.

The CMU/Stanford Packet Filter [62] introduced a *stack-based* virtual machine into the 4.3BSD kernel to interpret packet filters. To enable more efficient implementations, the Berkeley Packet Filter (BPF) [61] adopts a *register-based* virtual machine instead, which consists of two 32-bit registers and a scratch memory. BPF has gained a wide adoption in BSD and Linux kernels. Besides BPF, DTrace [12] and Lua on NetBSD [90] are two other in-kernel virtual machines.

A redesign of BPF in the Linux kernel started in 2014, first as an optimization of the internal representation of BPF instructions for 64-bit architectures [83]. It has since grown into a full RISC-like virtual machine, with 64-bit general-purpose registers, flexible control flow (e.g., bounded loops and BPF-to-BPF calls), and safe access to kernel memory. The generality and expressiveness have led to an explosion of tools and systems based on BPF, ranging from networking [38], security [79], tracing [35], to storage [7], virtualization [1, 71], and hardware offloading [43]. The new design is also called "extended BPF" or simply "BPF" in the Linux kernel, while the original design is referred to as *classic* BPF to avoid ambiguity. Unless otherwise noted, we follow this terminology and use BPF to refer to the new design. This paper focuses on building verified JITs for BPF.

More generally, the exokernels [26] demonstrate a diverse set of mechanisms for code downloading, such as accelerating packet filtering using JIT compilation [25], sandboxing machine code [92] using software-based fault isolation [77, 91], and analyzing file-system metadata using an in-kernel virtual machine [40]. Other extensibility mechanisms include using safe languages [6, 28, 55] and proof-carrying code [67].

Correctness of JIT compilation. Just-in-time compilation (JIT) is a well-studied dynamic code generation technique dating back to Lisp [3, 42] and regular expressions in the QED text editor [76, 87]. It has also been used for dynamically typed languages [14], emulators [5], and specialization [60, 73].

This paper considers JITs that are realized as *static* compilers, using static register allocation and performing no garbage collection for memory management. In contrast to sophisticated dynamic code generation systems such as those for Java or JavaScript, this simplicity makes static JITs applicable to a restricted environment such as the kernel [24].

There is a rich literature on compiler correctness. Readers may refer to Young [98] and Leroy [54] for overviews. Compilers, especially optimizing compilers, can have multiple intermediate representations and translation passes, whereas the JITs considered in this paper are much simpler and resemble a one-pass compiler. On the other hand, compilers usually

output assembly code, relying on a separate assembler and linker (e.g., GNU `as` and `ld`) to produce final machine code. The JITs run in the kernel and directly produce machine code, effectively combining a compiler, assembler, and linker.

The closest efforts in this area are the verified JITs by Myreen [65] and Jitk [94]. The former translates code in a simple stack-based instruction set to x86-32 (see §7), and is verified using the HOL4 theorem prover [80]. The JIT is implemented in HOL4 and translated to x86-32 machine code by a separate compiler [66]. Jitk builds on the CompCert verified compiler [53] to translate classic BPF to assembly, and is verified using the Coq theorem prover [86]. The JIT is implemented in Coq and extracted to OCaml code; it runs in user space rather than in the kernel due to the dependency on the OCaml runtime, an assembler, and a linker. Both efforts employ clean-slate designs, require manual proofs, and do not have a C implementation. Jitterbug is inspired by these efforts and shares the goal of building verified JITs, but prioritizes applicability to existing systems, proof automation, and implementation that can be reviewed independent of verification.

Compiler testing and fuzzing tools employ effective strategies to randomly generate input programs and check for miscompilation [58]. Csmith [97] and EMI fuzzers [52] have been used to find hundreds of bugs in GCC and LLVM. Kernel fuzzers such as syzkaller and trinity support generation of random BPF programs [23]. Serval [68] implements a bug finder for the compilation of BPF arithmetic and bitwise instructions. These tools generally do not exhaust all execution paths, thus providing no correctness guarantees for JITs.

Designing verified systems for deployment. Deployability is a desirable goal for formally verified systems, but it requires navigating an extra set of design trade-offs. As the first verified general-purpose microkernel, seL4 [46] pioneered many aspects of the design and deployment processes. For instance, it introduced a Haskell prototype as the bridge between formal methods and kernel developers, separating verification artifacts from the C implementation [45]. It has been deployed as a hypervisor to retrofit unverified, legacy software to power safety-critical systems [37, 47]. Another example is CompCert, the first verified C compiler. It has been integrated into the development process of control software for safety-critical systems [41, 53], replacing unverified compilers that were configured to disable optimizations due to risk concerns.

Cryptographic libraries are an attractive target for verification due to their essential role in security. For example, verified code from EverCrypt/HACL* [75, 99] and Fiat-Crypto [27] is used by Mozilla and Google, respectively. Amazon's s2n TLS implementation [16] is verified via a combination of manual and automated proofs.

Jitterbug presents a case study in applying formal methods to the BPF JITs in the Linux kernel. It shares these design challenges and addresses them with a precise specification and a proof strategy that scales to practical JIT implementations.

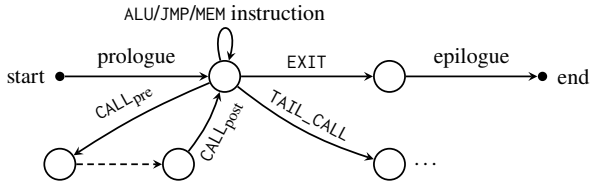


Figure 1: Transitions during the execution of a BPF program.

3 Case study

This section presents a brief overview of BPF and a case study of the BPF JIT bugs in the Linux kernel, which helped motivate the design of Jitterbug.

3.1 An overview of BPF

The BPF virtual machine consists of 12 explicit 64-bit registers: general-purpose registers `R0`–`R9`, a frame pointer `R10` that points to a stack memory region, and an internal register `AX` used by the kernel for rewrites (e.g., constant blinding against JIT spraying attacks [8]). It maintains a program counter `PC` and a tail-call counter `TCC`; the latter bounds the number of tail calls (to another BPF program without returning).

Currently, there are a total of 115 instruction opcodes, which can be categorized into the following:

- `ALU` (arithmetic and bitwise) instructions,
- `JMP` (unconditional and conditional jump) instructions,
- `MEM` (1-, 2-, 4-, and 8-byte memory access, and 4- and 8-byte atomic exchange-and-add) instructions,
- `CALL` to a kernel function or another BPF program; and
- `TAIL_CALL` and `EXIT`, which transfer control to another BPF program and the kernel, respectively.

Figure 1 depicts the execution of a BPF program. The input to a BPF program is provided by the kernel. Prologue and epilogue refer to initialization and cleanup code, respectively, for bridging the kernel. The BPF calling convention specifies that `R0` holds the return value, `R1`–`R5` pass arguments, and `R6`–`R9` are preserved across the call.

User processes may share data with BPF programs by creating BPF *maps* in the kernel, which are key/value stores of different data types. Maps may be accessed concurrently by BPF programs and user processes. Though there have been discussions, BPF has so far chosen not to specify a memory consistency model to avoid performance penalties [19].

Each BPF program consists of a sequence of instructions in *bytecode* (GCC/LLVM can compile C code to BPF). Upon receiving a BPF program from user space, the kernel invokes a checker to analyze whether the program is safe (e.g., free of division by zero, unbounded loops, and uninitialized register accesses) [34]; we refer to it as the BPF *checker* (rather than "BPF verifier" as by the Linux kernel to avoid ambiguity). If the BPF checker deems the program safe, the kernel invokes the JIT for compilation and attaches the resulting machine

```
/* rd[0]: upper 32 bits of the destination register
   rd[1]: lower 32 bits of the destination register
   tmp2[1]: a temporary register */
if (val < 32) {
  /* tmp2[1] = rd[1] >> val */
  emit(ARM_MOV_SI(tmp2[1], rd[1], SRTYPE_LSR, val), ctx);
  /* rd[1] = tmp2[1] | (rd[0] << (32 - val)) */
  emit(ARM_ORR_SI(rd[1], tmp2[1], rd[0], SRTYPE_ASL,
                  32 - val), ctx);
  /* rd[0] = rd[0] >> val */
  emit(ARM_MOV_SI(rd[0], rd[0], SRTYPE_LSR, val), ctx);
} else if (val == 32) {
  /* rd[1] = rd[0] */
  emit(ARM_MOV_R(rd[1], rd[0]), ctx);
  /* rd[0] = 0 */
  emit(ARM_MOV_I(rd[0], 0), ctx);
} else {
  /* rd[1] = rd[0] >> (val - 32) */
  emit(ARM_MOV_SI(rd[1], rd[0], SRTYPE_LSR,
                  val - 32), ctx);
  /* rd[0] = 0 */
  emit(ARM_MOV_I(rd[0], 0), ctx);
}
```

Figure 2: Incorrect result with zero `val` for `RSH64_IMM` (Arm32).

instructions to various hook points in the kernel for execution; otherwise, the kernel rejects the program. The JIT therefore considers safe programs only.

3.2 Bugs in BPF JITs

We manually inspected every commit to the BPF JITs in the Linux kernel from May 2014 (when the new BPF design was introduced) to April 2020, and categorized those that fixed JIT correctness bugs for Arm32, Arm64, RV64, x86-32, and x86-64; those for RV32 will be discussed in §7. We consider "correctness bugs" as JITs producing erroneous machine instructions, and exclude non-correctness bugs (e.g., memory leaks during JIT compilation) from the study. In total, there are 41 commits that fixed 82 JIT correctness bugs during this period. See §A.2 for a complete list.

Below we describe some representative bugs we have found using Jitterbug. These bugs are difficult to find even for veteran developers, and were not caught by the existing test suite. They can lead to security vulnerabilities, since the resulting machine instructions run in the kernel and may process input from untrusted sources. For clarity, BPF instructions and registers are in uppercase, while target machine ones are in lowercase.

Subtle architectural semantics. Figure 2 shows an excerpt of the Arm32 JIT for `RSH64_IMM`, the BPF logical right shift instruction of a 64-bit register by an immediate. Since the target architecture is 32-bit, the JIT uses two machine registers, represented by `rd[0]` and `rd[1]`, to hold the upper and lower 32 bits of a 64-bit BPF register, respectively. The BPF checker ensures that the shift amount `val` is within the range $[0, 63]$. The emitted instructions work as follows:

- when the shift amount `val` is less than 32, the result of the upper half is simply `rd[0] >> val`, and the result of the lower half is `rd[1] >> val` combined with the bits shifted from the upper half, `rd[0] << (32 - val)`;

```
/* check if rvoff is in the range [-2^31, 2^31 - 1] */
if (!is_32b_int(rvoff))
  return -ERANGE;
...
s64 upper = (rvoff + (1 << 11)) >> 12;
s64 lower = rvoff & 0xfff;
/* aupic t1,upper */
emit(rv_auipc(RV_REG_T1, upper), ctx);
/* jalr ra,lower(t1) */
emit(rv_jalr(RV_REG_RA, RV_REG_T1, lower), ctx);
```

Figure 3: Incorrect range check on `rvoff` for `CALL` (RV64).

- the result of the upper half is simply zero, as all the bits are shifted out, and the result of the lower half holds the bits shifted from the upper half.

One subtlety in Arm32 is that a zero immediate in the `lsr` (logical shift right) instruction means right-shift by 32 bits (i.e., shifting all bits out) [2: §F5.1.103]. Therefore, when the shift amount `val` is zero, the instructions produced by the JIT incorrectly set the destination register to zero, instead of behaving as a no-op. This is further complicated by inconsistent semantics in Arm32: a zero immediate in the shift left instruction means a no-op. We fixed the bug by changing the JIT to emit no instructions when `val` is zero.

Figure 3 shows another subtle bug in the RV64 JIT. Using a pair of `auipc+jalr` instructions is a standard way to support pc-relative call with a 32-bit offset on RISC-V [96]:

- `auipc t1,imm20` appends 12 low-order zero bits to a 20-bit immediate, sign-extends the 32-bit value to 64 bits, adds the sign-extended value to the address of the instruction, and writes the result in register `t1`;
- `jalr ra,imm12(t1)` jumps to a target address obtained by adding a sign-extended 12-bit immediate to the register `t1` and clearing the least-significant bit of the result for alignment; the address of the instruction following `jalr` is written to register `ra`.

One misconception is that `auipc+jalr` can reach any 32-bit offset in the range $[-2^{31}, 2^{31}-1]$ on 64-bit RISC-V (RV64), by using certain `imm20` and `imm12` values. Part of the confusion stems from the "RV32I base integer instruction set" chapter in the RISC-V instruction-set manual indicating that `auipc+jalr` "can jump anywhere in a 32-bit pc-relative address range." But the same does not hold on RV64: both `auipc` and `jalr` *sign-extend* their results to 64 bits, causing the reachable offset range to shift by -2^{11}. Therefore, the range check on `rvoff` in the JIT is incorrect, which can lead to an off-target jump.

Our report prompted the RISC-V instruction-set manual to add the following clarification: "Note that the set of address offsets that can be formed by pairing LUI with LD, AUIPC with JALR, etc. in RV64I is $[-2^{31}-2^{11}, 2^{31}-2^{11}-1]$." We fixed the bug in the JIT by using the clarified range for checking `rvoff`.

Subtle machine state. Figure 4 shows an excerpt of the x86-32 JIT for compiling BPF's `JSET64_REG` and `JSET32_REG` (in the form `BPF_JMP[32]|BPF_JSET|BPF_X` in C). The semantics of "`JSET64_REG DST,SRC,OFF`" is to perform a conditional

```
case BPF_JMP | BPF_JSET | BPF_X:
case BPF_JMP32 | BPF_JSET | BPF_X:
  bool is_jmp64 = BPF_CLASS(insn->code) == BPF_JMP;
  u8 dreg_lo = dstk ? IA32_EAX : dst_lo;
  u8 dreg_hi = dstk ? IA32_EDX : dst_hi;
  u8 sreg_lo = sstk ? IA32_ECX : src_lo;
  u8 sreg_hi = sstk ? IA32_EBX : src_hi;

  if (dstk) {
    EMIT3(0x8B, add_2reg(0x40, IA32_EBP, IA32_EAX),
          STACK_VAR(dst_lo));   /* eax <- dst_lo */
    if (is_jmp64)
      EMIT3(0x8B, add_2reg(0x40, IA32_EBP, IA32_EDX),
            STACK_VAR(dst_hi)); /* edx <- dst_hi */
  }
  if (sstk) {
    EMIT3(0x8B, add_2reg(0x40, IA32_EBP, IA32_ECX),
          STACK_VAR(src_lo));   /* ecx <- src_lo */
    if (is_jmp64)
      EMIT3(0x8B, add_2reg(0x40, IA32_EBP, IA32_EBX),
            STACK_VAR(src_hi)); /* ebx <- src_hi */
  }
  /* and dreg_lo,sreg_lo */
  EMIT2(0x23, add_2reg(0xC0, sreg_lo, dreg_lo));
  /* and dreg_hi,sreg_hi */
  EMIT2(0x23, add_2reg(0xC0, sreg_hi, dreg_hi));
  /*  or dreg_lo,dreg_hi */
  EMIT2(0x09, add_2reg(0xC0, dreg_lo, dreg_hi));
  goto emit_cond_jmp;   /* emit conditional jump */
```

Figure 4: Incorrect `eflags` value for `JSET32_REG` (x86-32).

jump when `DST&SRC` ("bitwise and" of two 64-bit BPF registers) is non-zero and fall through otherwise; the semantics of "`JSET32_REG DST,SRC,OFF`" is similar, using only the lower 32 bits of both `DST` and `SRC`.

Due to the limited number of registers on x86-32, the JIT spills some BPF registers on the stack. For simplicity, suppose that both `DST` and `SRC` are on the stack (i.e., both `dstk` and `sstk` are true). In this case, the JIT emits instructions to load the lower 32 bits of `DST` and `SRC` to `eax` and `ecx`, respectively. It also emits instructions to load the upper 32 bits to `edx` and `ebx` for `JSET64_REG`; the two registers are uninitialized for `JSET32_REG`.

One way to implement `JSET32_REG` is to emit a bitwise `and` of `eax` and `ecx`, followed by a conditional jump if the result is non-zero (i.e., the `zf` bit in the `eflags` register is clear). But the JIT emits extra `and` and `or` instructions that also use `edx` and `ebx`, which are uninitialized for `JSET32_REG`, incorrectly modifying `eflags`. The bug was not caught by the BPF selftests suite because none of the tests "polluted" `edx` and `ebx` with values that would cause the behavior to change. We fixed the bug by moving the last two `EMIT2` statements under a condition that `is_jmp64` is true.

There are other bugs in the excerpt: when `DST` is mapped to x86 registers and not spilled on the stack (i.e., `dstk` is false), the emitted instructions incorrectly clobber the registers, while the semantics of the BPF instructions requires `DST` not to change. We fixed the bugs by loading `DST` to `eax` and `ecx`, regardless of whether `DST` is on the stack.

Subtle instruction encoding. Below is an encoding bug in the x86-32 JIT for the BPF `LDXB` instruction, which loads a byte from memory. As its semantics requires the result to be zero-extended to 64 bits, the JIT attempts to emit "`mov dst_hi,0`" to clear the upper 32 bits, using the following C code:

```
EMIT3(0xC7, add_1reg(0xC0, dst_hi), 0);
```

Notice that `EMIT3` emits 3 bytes, but a correct "`mov dst_hi,0`" expects 6 bytes: the opcode `0xC7`, the ModR/M byte formed by `add_1reg(0xC0, dst_hi)`, followed by 4 bytes of zeros as the immediate. The consequence is not merely an incorrect `mov`: it also "swallows" 3 bytes from the next instruction, breaking the instruction stream and altering the meaning of the subsequent instructions. We fixed the bug by emitting "`xor dst_hi,dst_hi`" instead, which is also shorter (2 bytes).

3.3 Summary

Compared to the bugs in *classic* BPF JITs [15, 94], those in today's BPF JITs are more sophisticated due to the increased power of the BPF virtual machine. On the other hand, architecture-independent checks for BPF programs such as division by zero are now performed by the BPF checker, eliminating the need for the JITs to consider such cases.

While the Arm and RISC-V JITs emit instructions using well-defined macros (e.g., Figure 2) or functions (e.g., Figure 3), the x86 JITs directly emits raw bytes (e.g., Figure 4), partly due to the lack of a uniform instruction format on x86. Jitterbug therefore needs to model the semantics of their target architectures precisely; for x86, this means reasoning at the level of raw instruction bytes.

4 Specification

Jitterbug aims to rule out subtle bugs in BPF JITs through a formal specification, which is the focus of this section.

We begin with an intuitive description of what it means for a JIT to be correct. At a high level, running the machine code emitted by a JIT for a given source program should be equivalent to running a BPF interpreter with that source program. For example, both should compute the same return value and invoke the same kernel functions with the same arguments; any deviation indicates a bug. Jitterbug captures this intuition as a JIT correctness specification (§4.1).

Specifications like this are usually proved by induction, and the key to carrying out the proof is finding the right inductive invariant—a property preserved by the JIT translation of each individual source instruction. Inspired by the structure of the existing BPF JITs in the Linux kernel, Jitterbug introduces a *stepwise* specification that serves as our inductive invariant. As shown in Figure 5, this specification consists of a set of properties satisfied by individual translation steps, such as the generation of machine code for a single BPF instruction. Using the Lean theorem prover [22], we prove that any JIT that satisfies the stepwise specification implies our intuitive notion of correctness. This proof serves as the metatheory for

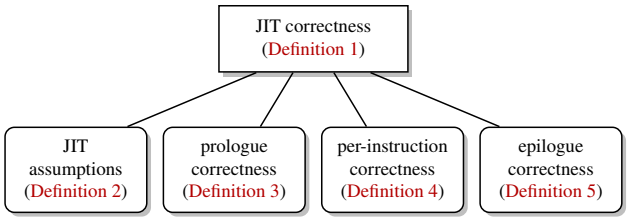


Figure 5: Jitterbug's stepwise specification (rounded-corner boxes) implies JIT correctness, shown by Theorem 1.

Jitterbug (§4.2). The stepwise specification itself is proved automatically for each JIT.

To illustrate how to apply the stepwise specification to prevent bugs, we use the BPF JIT for RV32 as an example. We also analyze alternative JIT implementations to demonstrate the generality of the specification (§4.3).

We end this section with a discussion of the limitations of Jitterbug's specification and how it relates to prior compiler correctness specifications (§4.4).

4.1 JIT correctness

Formalizing JIT correctness requires formalizing the behavior of the JIT, source BPF programs, and target machine programs, as follows.

First, we model a JIT as a function `JITCompile`. It takes a source program $code_S$ and JIT context ctx as input, and returns either a target program $code_T$ on success, denoted as $\texttt{JITCompile}(code_S, ctx) = code_T$; or $\bot$, indicating compilation error. Both source and target programs are represented as partial maps from addresses to instructions; some addresses may be unmapped. We define $code \subseteq code'$ to mean that any address that maps to some instruction in $code$ maps to the same instruction in $code'$.

The JIT context ctx is an implementation-defined data structure. It usually contains compiler configurations (e.g., the base address of the target program allocated by the kernel, denoted by $ctx[\texttt{base}]$) and analysis results of the source program, which are used by the JIT for code generation. We assume that the JIT context is *well-formed* with respect to the source program; this assumption is captured using a predicate $\texttt{wf}(code_S, ctx)$ specified by JIT developers. For example, one may specify that $ctx[\texttt{base}]$ is properly aligned.

Next, we model the execution of both source and target programs as abstract machines, described by a set of states Σ and a state transition function `step`. Given a state $\sigma \in \Sigma$, we write $\sigma[\cdot]$ to refer to a specific component of the state. For example, $\sigma[\texttt{pc}]$ is the value of the program counter.

The `step` function takes as input a state σ, a program $code$, and an oracle denoted by nd. The oracle nd is an infinite sequence of nondeterministically chosen bytes, which are used for modeling external interactions with the kernel (e.g., values loaded from BPF maps or returned by calls to kernel functions). Given these inputs, the `step` function produces the next state and a trace of externally visible *events* generated by executing the instruction at the program counter, $code[\sigma[\texttt{pc}]]$. The execution gets *stuck* if it triggers undefined behavior (e.g., the address $\sigma[\texttt{pc}]$ is unmapped in $code$). As shorthands, we write $\langle \sigma, code, nd\rangle \Longrightarrow \langle \sigma', tr\rangle$ to mean $\texttt{step}(\sigma, code, nd) = \langle \sigma', tr\rangle$, and $\langle \sigma, code, nd\rangle \Longrightarrow^* \langle \sigma', tr\rangle$ to mean that state σ' is reachable from zero or more applications of `step` starting from state σ, with concatenated trace tr.

The exact content of events is defined by each machine. For example, consider the BPF machine in Jitterbug. It defines the following events: $\texttt{load}(addr, val)$, $\texttt{store}(addr, val)$, $\texttt{call}(addr, args, val)$, `atomic_begin`, and `atomic_end`. It models each memory load as returning a fresh value provided by the oracle and producing a `load` event in the trace, since BPF maps may be modified outside the execution of a BPF program (§3.1). Each `step` may produce zero or more events. For example, the execution of `XADD32` (32-bit atomic exchange-and-add) produces `atomic_begin`, `load`, `store`, and `atomic_end`.

This model assumes read-only code, which prohibits JITs that produce self-modifying code [65]. It also assumes that the execution of a program is deterministic [53: §2.1], since the next state is uniquely determined by the current state, code, and oracle. Both assumptions match the BPF JITs in Linux.

In order to reason about the start and end of execution, each machine defines two predicates:

- $\texttt{initial}(x, ctx, \sigma)$, where σ is an initial state for input x and JIT context ctx; and
- $\texttt{final}(\sigma', v)$, where σ' is a final state with return value v.

Recall that the JIT considers only *safe* source programs. For example, the Linux kernel rejects BPF programs that the BPF checker deems unsafe (§3.1). We capture this guarantee with a predicate $\texttt{safe}(code)$, which specifies that executing $code$ always reaches a final state (i.e., the execution terminates without triggering any undefined behavior):

$$\forall\, x, \sigma, nd.\ \texttt{initial}(x, ctx, \sigma) \rightarrow \\ \exists\, \sigma', tr, v.\ \langle \sigma, code, nd\rangle \Longrightarrow^* \langle \sigma', tr\rangle \wedge \texttt{final}(\sigma', v).$$

In addition, since a target program generated by the JIT runs within the kernel, it must behave like a regular function and preserve the corresponding calling convention: for example, stack pointer and callee-saved registers must hold the same values before and after the execution. We capture these requirements in the *architectural safety* predicate $\mathcal{A}(\sigma_T, \sigma'_T)$, which constrains the initial and final values of all preserved target registers r to be the same, i.e., $\sigma_T[r] = \sigma'_T[r]$.

Using our model, we define JIT correctness as follows.

Definition 1 (JIT correctness)**.** A JIT is correct if for any safe source program $code_S$, well-formed JIT context ctx, and target program $code_T$ generated by the JIT such that $\texttt{safe}(code_S) \wedge \texttt{wf}(code_S, ctx) \wedge \texttt{JITCompile}(code_S, ctx) = code_T$, the following two conditions hold:

1. The execution of source program $code_S$ and that of target program $code_T$ produce the same trace and return value.

$$\forall\, x, \sigma_S, \sigma_T, nd, tr, v.$$
$$\texttt{initial}_S(x, ctx, \sigma_S) \wedge \texttt{initial}_T(x, ctx, \sigma_T) \rightarrow$$
$$\Big((\exists\, \sigma'_S.\ \langle \sigma_S, code_S, nd\rangle \Longrightarrow^* \langle \sigma'_S, tr\rangle \wedge \texttt{final}_S(\sigma'_S, v)) \leftrightarrow$$
$$(\exists\, \sigma'_T.\ \langle \sigma_T, code_T, nd\rangle \Longrightarrow^* \langle \sigma'_T, tr\rangle \wedge \texttt{final}_T(\sigma'_T, v))\Big).$$

2. Any final state reachable by executing target program $code_T$ satisfies architectural safety.

$$\forall\, x, \sigma_T, \sigma'_T, nd, tr, v.\ \texttt{initial}_T(x, ctx, \sigma_T) \wedge \texttt{final}_T(\sigma'_T, v) \wedge$$
$$\langle \sigma_T, code_T, nd\rangle \Longrightarrow^* \langle \sigma'_T, tr\rangle \rightarrow \mathcal{A}(\sigma_T, \sigma'_T).$$

The first property can be viewed as a *bisimulation* between source and target machines [54: §2]: the JIT produces a target program that preserves the behavior of the source program, and any behavior of the target program is permitted by the source program. Additionally, given that the source program is safe, this property implies that the target program produced by the JIT is safe (i.e., terminates without undefined behavior). The second property further requires the target program to correctly save and restore the corresponding architectural state. Both guarantees are critical for in-kernel execution.

4.2 Stepwise specification

Given Definition 1, our goal is to devise a stepwise specification (i.e., an inductive invariant) that both implies JIT correctness and is amenable to automated verification. We achieve this goal by imposing structure on the JIT compilation process so that we can reason about the correctness of individual compilation steps, as follows.

Inspired by the existing BPF JITs in the Linux kernel, we suppose that the JIT generates a target program in a *per-instruction* fashion. Specifically, the target program consists of machine instructions for the prologue, each source instruction, and the epilogue (Figure 1). We do not assume any particular code layout. For example, one may produce the target program sequentially:

```
code_T  = EmitPrologue(ctx)
for i in [0, |code_S| − 1]:
    code_T  += EmitInstruction(ctx, i, code_S[i])
code_T  += EmitEpilogue(ctx)
```

We formalize our assumptions about the JIT below.

Definition 2 (JIT assumptions)**.** We assume that for any safe source program $code_S$, well-formed JIT context ctx, and target program $code_T$ produced by a JIT such that $\texttt{safe}(code_S) \wedge \texttt{wf}(code_S, ctx) \wedge \texttt{JITCompile}(code_S, ctx) = code_T$, the target program $code_T$ contains the machine instructions produced by each translation step:

- $\exists\, p.\ \texttt{EmitPrologue}(ctx) = p \wedge p \subseteq code_T$.
- $\forall\, i, insn.\ code_S[i] = insn \rightarrow$
  $\exists\, p.\ \texttt{EmitInstruction}(ctx, i, insn) = p \wedge p \subseteq code_T$.
- $\exists\, p.\ \texttt{EmitEpilogue}(ctx) = p \wedge p \subseteq code_T$.

With these assumptions, the stepwise specification boils down to the correctness of each translation step: `EmitPrologue`, `EmitInstruction`, and `EmitEpilogue`. Jitterbug allows developers to provide two relations as invariants maintained by their JIT implementations:

- $\sigma_S \sim_{ctx} \sigma_T$ relates source state σ_S and target state σ_T with respect to JIT context ctx. For example, it may specify that the value of a BPF register in σ_S is equal to that of the machine register the JIT uses to realize the BPF register in σ_T.
- $\mathcal{I}_{ctx}(\sigma_{T_0}, \sigma_T)$ relates initial target state σ_{T_0} and non-final target state σ_T with respect to JIT context ctx. For example, the prologue usually saves callee-saved registers to a designated memory region; $\mathcal{I}_{ctx}$ may specify that the values of callee-saved registers in σ_{T_0} are equal to those in that region in σ_T.

Below we describe the correctness definition for each translation step. We denote the empty trace as ϵ.

Definition 3 (Prologue correctness)**.** A JIT emits a correct prologue if executing the prologue results in a target state that establishes the invariants, and produces an empty trace:

$$\forall\, code_S, ctx, p, x, \sigma_S, \sigma_T, nd.\ \texttt{wf}(code_S, ctx) \wedge$$
$$\texttt{EmitPrologue}(ctx) = p \wedge$$
$$\texttt{initial}_S(x, ctx, \sigma_S) \wedge \texttt{initial}_T(x, ctx, \sigma_T) \rightarrow$$
$$\exists\, \sigma'_T.\ \langle \sigma_T, p, nd\rangle \Longrightarrow^* \langle \sigma'_T, \epsilon\rangle \wedge (\sigma_S \sim_{ctx} \sigma'_T) \wedge \mathcal{I}_{ctx}(\sigma_T, \sigma'_T).$$

Definition 4 (Per-instruction correctness)**.** A JIT emits correct target instructions for a given source instruction if executing the emitted instructions results in a target state that preserves the invariants, and produces the same trace as executing the source instruction:

$$\forall\, code_S, ctx, i, insn, p, \sigma_S, \sigma_T, \sigma_{T_0}, nd, tr.\ \texttt{wf}(code_S, ctx) \wedge$$
$$code_S[i] = insn \wedge \sigma_S[\texttt{pc}] = i \wedge$$
$$\texttt{EmitInstruction}(ctx, i, insn) = p \wedge$$
$$\langle \sigma_S, code_S, nd\rangle \Longrightarrow \langle \sigma'_S, tr\rangle \wedge (\sigma_S \sim_{ctx} \sigma_T) \wedge \mathcal{I}_{ctx}(\sigma_{T_0}, \sigma_T) \rightarrow$$
$$\exists\, \sigma'_T.\ \langle \sigma_T, p, nd\rangle \Longrightarrow^* \langle \sigma'_T, tr\rangle \wedge (\sigma'_S \sim_{ctx} \sigma'_T) \wedge \mathcal{I}_{ctx}(\sigma_{T_0}, \sigma'_T).$$

Definition 5 (Epilogue correctness)**.** A JIT emits a correct epilogue if executing the epilogue results in a final target state that satisfies architectural safety, and produces the same return value as in the source final state and an empty trace:

$$\forall\, code_S, ctx, p, \sigma_S, v, \sigma_T, \sigma_{T_0}, nd.\ \texttt{wf}(code_S, ctx) \wedge$$
$$\texttt{EmitEpilogue}(ctx) = p \wedge$$
$$\texttt{final}_S(\sigma_S, v) \wedge (\sigma_S \sim_{ctx} \sigma_T) \wedge \mathcal{I}_{ctx}(\sigma_{T_0}, \sigma_T) \rightarrow$$
$$\exists\, \sigma'_T.\ \langle \sigma_T, p, nd\rangle \Longrightarrow^* \langle \sigma'_T, \epsilon\rangle \wedge \texttt{final}_T(\sigma'_T, v) \wedge \mathcal{A}(\sigma_{T_0}, \sigma'_T).$$

Together, these three properties imply JIT correctness given the JIT assumptions. We prove the following theorem in Lean:

Theorem 1 (Stepwise soundness)**.** JIT assumptions $\wedge$ prologue correctness $\wedge$ per-instruction correctness $\wedge$ epilogue correctness $\rightarrow$ JIT correctness.

With Theorem 1 as a metatheory, Jitterbug proves the correctness of a JIT implementation by proving the properties in

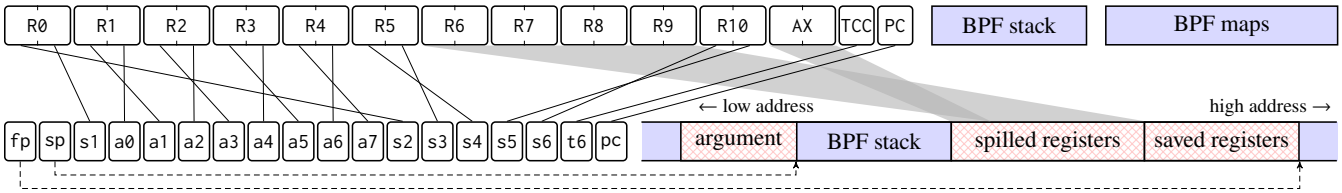


Figure 6: Mapping from BPF state (upper half) to RV32 state (lower half). Rounded-corner boxes denote registers and rectangular boxes denote memory. Shaded regions are memory accessible by BPF programs and crosshatched regions for internal use.

Definitions 3, 4, and 5 via automated verification (see §5). The JIT context well-formedness `wf` and assumptions are assumed to be correct and trusted. The invariants ($\sim_{ctx}$ and $\mathcal{I}_{ctx}$) are untrusted: if incorrect invariants are provided, verification fails.

4.3 Applying the stepwise specification

The stepwise specification is parameterized by assumptions (well-formedness of JIT context `wf`) and invariants ($\sim_{ctx}$ and $\mathcal{I}_{ctx}$), which reflect how JIT developers intend to establish correctness. We illustrate how to apply the stepwise specification to the BPF JIT for RV32 by specifying the assumptions and invariants regarding registers, program counters, and memory. We also describe how one may specify them for the alternative JIT implementations we have considered.

Figure 6 shows the design of the BPF JIT for RV32. The upper half denotes the BPF state, including registers (`R0`–`R10`, `AX`), counters (tail-call counter `TCC` and program counter `PC`), a stack memory region, and maps of shared data (§3.1). The lower half denotes the RV32 state, including registers (`fp`, `sp`, `a0`–`a7`, `s1`–`s6`, `t6`; those not mapped to BPF registers are omitted), a machine program counter `pc`, and memory.

Registers. Since BPF registers are 64-bit and RV32 is a 32-bit architecture, the JIT realizes each BPF register using either a pair of RV32 registers (e.g., `R1` using `a0` and `a1`) or 64 bits in the "spilled registers" memory region (e.g., `R6`). This register mapping is static and pre-determined, eliminating the need for register allocation at compilation time. Other BPF JITs in the Linux kernel use similar register mappings.

The register mapping is handcrafted to achieve good performance. For instance, recall that BPF designates `R1`–`R5` to pass function-call arguments, while the RISC-V calling convention uses `a0`–`a7`, plus the stack if needed [20]. To minimize register save and restore, the JIT realizes `R1`–`R4` using `a0`–`a7`. For `R5`, the JIT emits instructions to push the corresponding `s3`, `s4` to the "argument" memory region before the call.

To specify the relation $\sigma_S \sim_{ctx} \sigma_T$ between source and target states, let $\varphi_{\text{reg}}(ctx, \sigma_T, r)$ denote the value stored at the target location(s) to which a BPF register r is mapped (e.g., `R1` mapped to `a0`, `a1`) with respect to JIT context ctx. A strawman approach is to require a strict equivalence: $\sigma_S[r] = \varphi_{\text{reg}}(ctx, \sigma_T, r)$ for every BPF register r. With this relation, the stepwise specification would require that if every BPF register and the mapped locations contain equivalent values initially, their values remain equivalent after executing a BPF instruction and the emitted machine instructions, respectively. One such example is the *partial* specification used by the BPF bug finder in Serval [68: §7]; the specification is partial because it does not support reasoning about control flow (e.g., program counters) or memory and cannot be used to prove JIT correctness.

While it is useful for finding bugs, the strawman relation is too restrictive for verification. First, if a BPF program does not use a certain register, it should be safe for the JIT to skip emitting code for initializing the corresponding target locations, but doing so violates the strict equivalence. Second, the relation is difficult to establish in the presence of calls. To see why, consider the BPF register `R1`, which is *not* preserved across a BPF `CALL` instruction (§3.1). `R1` is thus considered *uninitialized* after the call as per the BPF semantics (the BPF checker ensures that `R1` will be written to before any further use). On the other hand, `R1` is mapped to `a0`, `a1`, both of which hold the return value after the call as per the RISC-V calling convention (the JIT emits instructions to further copy their values to `s1`, `s2` to match the BPF calling convention for `R0`). Therefore, `R1` and the corresponding `a0`, `a1` do not hold equivalent values after the call, which violates the strict equivalence.

To relax the strict equivalence and give the JIT more freedom regarding uninitialized BPF registers, we augment the state of the BPF machine with an `initialized` set, which represents the set of registers that are initialized at this point; the set is updated based on the semantics of each BPF instruction. For example, `DST` is added to the set after "`MOV64_IMM DST, IMM`," as it is written to by the instruction. Similarly, `R1`–`R5` are removed from the set after `CALL`, as they are not preserved across the call and become uninitialized. In doing so, it suffices to require equivalence $\sigma_S[r] = \varphi_{\text{reg}}(ctx, \sigma_T, r)$ for every BPF register $r \in \sigma_S[\texttt{initialized}]$, effectively excluding uninitialized ones.

Program counters. Let $\varphi_{\text{pc}}(ctx, i)$ denote the target address to which the i-th BPF instruction is mapped in JIT context ctx. This is useful for a JIT to implement the compilation of jump instructions. It also allows us to relate program counters in BPF and machine states as an invariant $\varphi_{\text{pc}}(ctx, \sigma_S[\texttt{pc}]) = \sigma_T[\texttt{pc}]$.

To define φ_{pc}, one simple approach is to require the JIT to emit a *fixed* number of machine instructions for each BPF instruction (e.g., by padding with NOPs) [33]. In this case, we have $\varphi_{\text{pc}}(ctx, i) = ctx[\texttt{base}] + i \times N$, where $ctx[\texttt{base}]$

is the starting address of the emitted machine instructions determined by JIT context and N is a pre-determined number of machine instructions large enough to compile any BPF instruction. This is simple to specify and implement, but the emitted code wastes space and CPU cycles.

A more efficient approach is to emit a variable number of machine instructions for each BPF instruction. For example, the BPF JITs in the Linux kernel maintain an *offset table* in the JIT context to map each BPF instruction index to an offset into the emitted code; in this case $\varphi_{\text{pc}}(ctx, i)$ is defined by simply consulting the offset table. The JITs construct the offset table by repeating the compilation process until the table converges, or fail if an upper bound on the number of iterations is reached (e.g., 16 in the BPF JIT for RV32).

For flexibility, we choose not to specify how to construct the offset table in the JIT context. Instead, we specify the property a valid JIT context should satisfy. A key observation is that such JITs emit *consecutive* blocks of machine instructions, one block for each BPF instruction. As a result, the difference between the target addresses for a BPF instruction $code_S[i]$ and its successor $code_S[i+1]$ must be equal to the number of bytes emitted for $code_S[i]$. We capture the observation using the well-formedness predicate `wf` over source program $code_S$ and JIT context ctx for any i-th BPF instruction:

$$\texttt{EmitInstruction}(ctx, i, code_S[i]) = p \rightarrow$$
$$|p| = \varphi_{\text{pc}}(ctx, i+1) - \varphi_{\text{pc}}(ctx, i).$$

Here $|p|$ denotes the length of machine instructions p (in bytes). This allows for both NOP-padding and the more sophisticated JIT implementations such as those in the Linux kernel. Note that this is an assumption on the validity of the JIT context, which does *not* rule out bugs in the construction of the offset table (see §4.4). A JIT may validate the offset table by checking that this predicate holds at compilation time.

Memory. One approach to relating the memory state of source and target machines, denoted by $\sigma_S[\texttt{mem}]$ and $\sigma_T[\texttt{mem}]$, respectively, is to require $\sigma_S[\texttt{mem}](a) = \sigma_T[\texttt{mem}](a)$ for every address a [65], where memory is modeled as a map from addresses to values. But this approach assumes a closed system (see §7 for such a JIT) and does not fit BPF. For example, both user processes and other BPF programs may concurrently modify memory to which a BPF program has access; therefore, consecutive loads from the same address in the BPF program may return different values. A further complication is that BPF does not specify a memory consistency model (§3.1), effectively assuming that of the underlying architecture.

We observe that a BPF JIT does not need to reason about the behavior of concurrent memory accesses [13, 56]. Instead, the goal is to faithfully translate BPF memory accesses to ones in the target machine, which is a simpler task. Based on this observation, we employ a hybrid approach to specify the invariants for BPF JITs using traces and maps, as follows.

Each target machine models memory as consisting of two disjoint parts, one corresponding to shared memory and the other for internal use (Figure 6). The memory layout used by a JIT determines which target addresses are shared and which are internal. The internal memory is simply a map from addresses to values, since it is private to each execution and the effects are not externally visible. The shared memory captures memory-related effects using events in a trace (§4.1). Since the BPF machine adopts the memory model of the underlying architecture, Jitterbug relates the traces of the BPF and target machines by using the same memory model for both; i.e., the BPF and target traces are drawn from the same set of all possible memory events. Given this correspondence, it suffices to require the traces produced by the BPF and target machines to be identical.

The requirement of having identical traces suffices for the BPF JITs. One exception is that older versions of the BPF JIT for Arm64 use Arm's exclusive access instructions in a busy loop [2: §B2], which violates the requirement. Newer versions of the JIT have switched to using atomic instructions, which satisfies the requirement. We decide not to relax the requirement of having identical traces to keep the specification simple.

4.4 Discussion and limitations

Jitterbug's JIT correctness (Definition 1), especially the use of traces, is inspired by the specification of CompCert [54]. Jitterbug's specification differs in the following ways. First, in-kernel execution imposes stricter requirements on the source program (e.g., determinism, termination, and absence of undefined behavior), allowing us to prove stronger properties. Second, Jitterbug uses fine-grained models of target architectures to precisely reason about low-level state (e.g., program counter and stack pointer), whereas CompCert uses a more abstract semantics for assembly [64: §5] and relies on a separate assembler and linker. Third, the per-instruction compilation process of such JITs enables us to develop a stepwise specification amenable to automated verification.

Jitterbug trusts the correctness of the assumptions (§4.2). Therefore, it cannot catch bugs in the JIT context (e.g., offset-table construction) or layout of the target program. We manually examine the existing BPF JIT correctness bugs in the Linux kernel (§3.2), and determine that out of the 82 bugs, the specification can catch all but two bugs, both in offset-table construction. This shows the effectiveness of the specification.

Jitterbug's specification permits "null" JIT implementations that fail on all source programs; we use existing test suites (e.g., the BPF selftests) to assess the feature completeness of JITs. It focuses on the JIT and cannot rule out bugs in the BPF checker, memory management for code images, or how the kernel uses the JIT. It does not model the instruction cache or memory permissions, relying on the kernel to correctly flush the cache and set up permissions. It does not provide any guarantees against microarchitectural timing channels [32, 48].

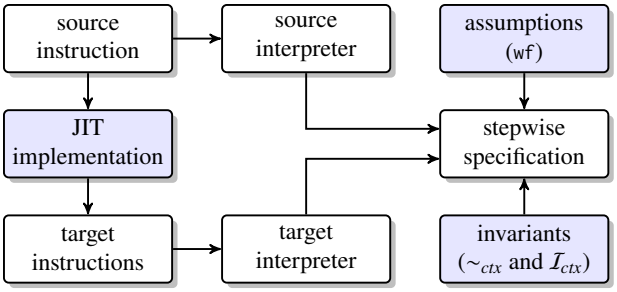


Figure 7: Jitterbug's verification pipeline. Shaded boxes denote inputs provided by JIT developers.

5 Proving JIT correctness

Jitterbug extends automated verification to JIT correctness, a form of compiler correctness tailored for in-kernel execution. This section describes how Jitterbug achieves the automation.

As shown in Figure 7, Jitterbug provides the stepwise specification and the executable semantics (i.e., interpreters) for BPF and various architectures. It asks JIT developers for a JIT implementation, and assumptions and invariants regarding the implementation. All the inputs are written in the Rosette language (the JIT is written in the DSL described in §6).

Jitterbug builds on Serval for automated verification [68]. It invokes Rosette to reduce all the inputs to symbolic constraints via symbolic evaluation, and an SMT solver to check the satisfiability of these constraints. For symbolic evaluation to terminate, the JIT implementation and the execution of both interpreters must be free of unbounded loops [88]. The BPF JITs satisfy this requirement.

Below we highlight three key challenges in automated verification of JITs and how Jitterbug addresses these challenges.

Instantiation of existential quantifiers. To prove the stepwise specification, Jitterbug has to construct *some* execution of target instructions emitted by the JIT and show that the execution exhibits the same behavior as that of a source instruction. Automating the construction is challenging.

To see why, consider the specification for per-instruction correctness (§4.2). Letting $\vec{x}$ stand for the universally quantified variables in Definition 4, the specification says that the target machine executes some finite number of steps, k, to produce a state σ'_T that satisfies the inductive invariant P. Making the number of steps k explicit, we can write the correctness formula as $\forall \vec{x}.\, \exists k.\, P(\vec{x}, k)$, or equivalently, $\exists f.\, \forall \vec{x}.\, P(\vec{x}, f(\vec{x}))$, where f is a Skolem function that computes the right k for each combination of the variables $\vec{x}$. The verification problem that Jitterbug solves therefore involves constructing f. In other words, Jitterbug must determine the number of steps to run symbolic evaluation with emitted instructions, and this value $f(\vec{x})$ may depend on the source program, JIT context, source and target states, etc.

In a restricted setting where the JIT emits straight-line code without any branches, $f(\vec{x})$ is simply the number of emitted instructions. The BPF bug finder in Serval and synthesis-based superoptimizers [72] all assume this setting and use the corresponding basic realization of f. But Jitterbug considers JITs that can emit code with branches, and when executing such code, the target machine can take a different number of steps depending on the input state. This rules out the straightforward realization of f that counts the number of emitted instructions.

To illustrate the challenge of computing f in our setting, consider the instructions emitted by the RV32 JIT for the BPF instruction "`JNE64_REG DST,SRC,OFF`" (jump to offset `OFF` if the values in `DST` and `SRC` differ). The JIT may emit different blocks of RV32 instructions, conditioned on whether it spills the registers (requiring `lw` to load from stack) or the offset requires a far jump (`jal` or `auipc+jalr`). Figure 8 shows three examples of these blocks and the $f(\vec{x})$ values for executing them, which vary depending on the register state and instructions. In general, constructing f requires human insight [63, 98], so Jitterbug allows JIT developers to provide a manually constructed f. In practice, however, Jitterbug can automate the construction of f for BPF JITs as follows.

To compute f, Jitterbug requires the target interpreter to maintain the symbolic program counter in the form `base` + *offset*, where `base` is the (symbolic) starting address of instructions. Maintaining this form is straightforward for most instructions. For instructions with subtler semantics, the interpreter achieves this by rewriting the program counter via *symbolic optimization* [68, 74]. For example, RISC-V's `jalr` sets the least-significant bit of the program counter to zero (§3.2), causing it to take the form (`base` + *offset*) `&` *mask*. The interpreter rewrites this expression by dropping the mask and checking that the resulting expression is equivalent (i.e., that the program counter is properly aligned).

Given a program counter of the form `base` + *offset*, Jitterbug provides a reusable procedure for constructing f through symbolic evaluation. It extracts the offset from the program counter expression and applies a simple rule: stop symbolic evaluation either if the offset is concrete but leaves the block of emitted instructions, or if the offset becomes symbolic. The intuition is that branching with a symbolic offset likely leaves the block, because the JIT generally produces such branching instructions by consulting the offset table in the JIT context (§4.3), which is considered symbolic for verification. Internal branching tends to have a concrete offset, for which symbolic evaluation continues.

This procedure guesses an f for verifying that the target machine reaches a desired state after taking $f(\vec{x})$ steps. It does not guarantee to find the right f if one exists, though it is sufficient for all the JITs we have studied and works well in practice. The procedure is untrusted: choosing a wrong f causes the target machine to either get stuck or enter a state that violates the inductive invariant, but it does not cause an erroneous JIT implementation to pass verification.

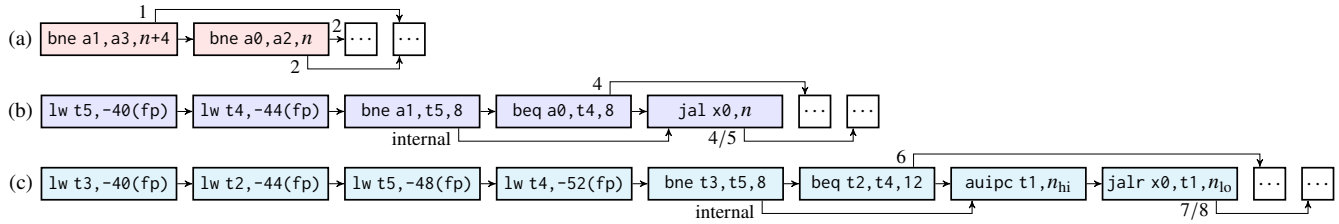


Figure 8: Emitted RV32 instruction blocks for BPF's `JNE64_REG` with registers (a) `R1`, `R2`; (b) `R1`, `R6`; and (c) `R6`, `R7` and with an offset of different ranges. `R1` and `R2` are realized using RV32 registers, and `R6` and `R7` are spilled on the stack (Figure 6). Straight and elbow arrows denote falling through and branching, respectively; those leaving the blocks are labeled with values of $f(\vec{x})$.

Symbolic evaluation of symbolic instructions. As shown in Figure 8, the JIT may emit different blocks of target instructions for a given source instruction opcode. When Jitterbug symbolically evaluates a JIT implementation, it produces a symbolic representation of all of these instruction blocks. This representation takes the form of *symbolic* instructions that may contain symbolic values in register and immediate fields. To verify the JIT, Jitterbug must then evaluate the target interpreter on both a symbolic input state *and* a symbolic program. This is in contrast to prior work on verifying systems code such as Serval, where the input state is symbolic but the program itself is concrete (e.g., all register and immediate fields are concrete bytes). Reasoning about symbolic programs both magnifies existing challenges to scaling verification and creates new ones. We discuss one example of each.

The first challenge is path explosion. While common to all tools based on symbolic evaluation, this problem becomes exponentially worse in the presence of symbolic code. For example, the BPF JIT for RV64 compiles `LD64_IMM` to a variable number of instructions to load a 64-bit immediate in chunks, selecting each instruction based on the chunk value and destination register. This amounts to reasoning about a total of 2,181 types of blocks of RV64 instructions for downstream stages, out of which 307 are feasible, applied to all possible input instructions (roughly, 2^{64}). *Symbolic execution* [17, 44], which explores individual paths separately, is thus not a good fit for this verification pipeline.

Jitterbug instead adopts Rosette's strategy for symbolic evaluation [89: §4] to merge the program state at each control-flow join, but it forces a split on every possible (concrete) opcode of symbolic instructions. The intuition is that both the JIT and interpreters tend to handle each opcode separately; splitting on the opcode enables opportunities for concrete evaluation. This strategy works well in practice: it avoids path explosion and leads to easier-to-solve constraints.

The second challenge is that Jitterbug interpreters, unlike those in Serval, must be designed to work on both symbolic state and symbolic instructions. Failing to do so both causes state explosion and produces constraints difficult for SMT solving. For example, the interpreters in Serval represent the CPU state using a vector of bitvectors (one bitvector per register), and encode accessing register r_i as indexing into the vector using integer i. This is suitable for concrete instructions, where i is concrete and the generated constraints are restricted to the theory of bitvectors. But with symbolic instructions, a symbolic register index i causes symbolic evaluation to produce constraints that also use the theory of (mathematical) integers. Mixing integers and bitvectors is expensive for solving and can lead to verification bottlenecks [39: §3].

We develop interpreters that account for symbolic instructions and thus can work with Jitterbug. For example, we carefully avoid integers in instruction semantics to restrict resulting constraints to the theories of equality with uninterpreted functions and bitvectors, a decidable fragment of first-order logic. Additionally, recall that the BPF JITs for x86 emit raw bytes (§3.2) and thus require a decoder for verification. We implement an x86 decoder that works on symbolic bytes. The development process is guided by using symbolic profiling to identify verification performance bottlenecks [10] and applying symbolic optimization to fine-tune symbolic evaluation [68: §4].

Axiomatization of expensive SMT operations. Both BPF and machine interpreters provide arithmetic instructions for multiplication, division, and remainder. Reasoning about these operations is expensive for SMT solvers [4, 49], and has been a source of timeouts in verification practice [30, 36].

To avoid such expensive reasoning, Jitterbug takes a standard *axiomatization* approach [50: §3.2] by replacing these bitvector operations with uninterpreted functions $\texttt{mul}_n$, $\texttt{div}_n$, and $\texttt{rem}_n$ ($n = 32, 64$) in instruction semantics. For example, the BPF JIT for RV32 translates BPF's `DIV32_REG` into using RISC-V's `divu`; both instructions are encoded using uninterpreted function $\texttt{div}_{32}$ (with variations for handling division by zero). Proving the correctness of this translation does *not* require the semantics of division, thereby scaling verification.

This approach is less general than using SMT's built-in bitvector operations, because it ignores the semantics of these operations and might reject valid JIT implementations. For example, the JIT may reorder the operands of a multiplication in emitted instructions; for target architectures lacking native instructions for remainder or 64-bit multiplication, the JIT may emit instructions that emulate the behavior. Proving such a JIT correct requires additional properties about the operations.

Jitterbug captures these properties using the following axioms, which are sufficient to verify all the JITs we have studied:

- commutativity of `mul`: $\texttt{mul}_n(x,y) = \texttt{mul}_n(y,x)$;
- remainder: $\texttt{rem}_n(x,y) = x - \texttt{mul}_n(\texttt{div}_n(x,y),y)$;
- commutativity of `mulhu`: $\texttt{mulhu}_n(x,y) = \texttt{mulhu}_n(y,x)$; and
- decomposition of $\texttt{mul}_{64}$: $\texttt{mul}_{64}(x,y) = (\texttt{mulhu}_{32}(x_{\text{lo}},y_{\text{lo}}) + \texttt{mul}_{32}(x_{\text{hi}},y_{\text{lo}}) + \texttt{mul}_{32}(x_{\text{lo}},y_{\text{hi}})) \oplus \texttt{mul}_{32}(x_{\text{lo}},y_{\text{lo}})$.

Here x and y are bitvectors; subscripts "lo" and "hi" denote the lower and upper half bits, respectively; $\oplus$ denotes bitvector concatenation; and `mulhu` is an auxiliary uninterpreted function for modeling the upper bits of a product. For example, x86's 32-bit unsigned multiplication instruction stores a 64-bit product in registers `edx` and `eax`; the x86 interpreter encodes their values using $\texttt{mulhu}_{32}$ and $\texttt{mul}_{32}$, respectively.

These axioms are shared by the verification of the BPF JITs across architectures. As a sanity check, we formalize and manually prove them using Lean [22].

6 Implementing a JIT

DSL. Figure 9 shows an excerpt of the BPF JIT for RV32, written in the Jitterbug DSL. The DSL is implemented as a Rosette library and reflects a structured subset of C: booleans, (machine) integers, array accesses ("`@`"), as well as conditional and switch statements. This subset is minimal and sufficient to support the development of the BPF JIT for RV32. It does not support address-of, dereference, or unstructured constructs (e.g., `goto` or fallthrough in `switch`).

Jitterbug extracts the final C code from JIT fragments written in the DSL, a code template (including glue code not covered by the DSL), and a type mapping (not shown here; both array accesses to `bpf2rv32` and calls to `bpf_get_reg64` return a value of type "`const s8 *`"). Jitterbug does not perform any type checking for the C code.

Using the DSL simplifies verification by avoiding the need to model the C semantics. One can also "escape" from the DSL to use the full Rosette language, though in that case Jitterbug is unable to perform C code extraction; we leverage this to simplify porting the existing BPF JITs from C to Jitterbug.

Synthesis. As an application of Jitterbug's specification and verification, we use Rosette's support for program synthesis to optimize the BPF JIT for RV32 [59]. We do so by synthesizing JIT fragments written in (a subset of) the DSL, where each fragment takes as input a BPF instruction with a given opcode (e.g., `ADD64_REG`) and emits a short sequence of RV32 instructions with equivalent behavior. We use the standard approach of writing program sketches [11, 82] to compactly define a space of JIT fragments for compiling `ALU` instructions. The synthesizer searches this space for the shortest fragment that satisfies per-instruction correctness (Definition 4), according to the Jitterbug verifier.

```
(func (emit_alu_r64 dst src ctx op)
  (var [tmp1 (@ bpf2rv32 TMP_REG_1)]
       [tmp2 (@ bpf2rv32 TMP_REG_2)]
       [rd   (bpf_get_reg64 dst tmp1 ctx)]
       [rs   (bpf_get_reg64 src tmp2 ctx)])
  (switch op
    [(BPF_ADD)
      (cond
        [(equal? rd rs)
          (emit (rv_srli RV_REG_T0 (lo rd) 31) ctx)
          (emit (rv_slli (hi rd) (hi rd) 1) ctx)
          (emit (rv_or (hi rd) RV_REG_T0 (hi rd)) ctx)
          (emit (rv_slli (lo rd) (lo rd) 1) ctx)]
        [else
          (emit (rv_add (lo rd) (lo rd) (lo rs)) ctx)
          (emit (rv_sltu RV_REG_T0 (lo rd) (lo rs)) ctx)
          (emit (rv_add (hi rd) (hi rd) (hi rs)) ctx)
          (emit (rv_add (hi rd) (hi rd) RV_REG_T0) ctx)])]
    ...))
```

(a) JIT implementation written in the DSL.

```
void emit_alu_r64(const s8 *dst, const s8 *src,
                  struct rv_jit_context *ctx, const u8 op)
{
// clang-format on
@|emit_alu_r64|
// clang-format off
}
```

(b) C code template, where `@|...|` expands to generated code.

```
void emit_alu_r64(const s8 *dst, const s8 *src,
                  struct rv_jit_context *ctx, const u8 op)
{
  const s8 *tmp1 = bpf2rv32[TMP_REG_1];
  const s8 *tmp2 = bpf2rv32[TMP_REG_2];
  const s8 *rd = bpf_get_reg64(dst, tmp1, ctx);
  const s8 *rs = bpf_get_reg64(src, tmp2, ctx);

  switch (op) {
  case BPF_ADD:
    if (rd == rs) {
      emit(rv_srli(RV_REG_T0, lo(rd), 31), ctx);
      emit(rv_slli(hi(rd), hi(rd), 1), ctx);
      emit(rv_or(hi(rd), RV_REG_T0, hi(rd)), ctx);
      emit(rv_slli(lo(rd), lo(rd), 1), ctx);
    } else {
      emit(rv_add(lo(rd), lo(rd), lo(rs)), ctx);
      emit(rv_sltu(RV_REG_T0, lo(rd), lo(rs)), ctx);
      emit(rv_add(hi(rd), hi(rd), hi(rs)), ctx);
      emit(rv_add(hi(rd), hi(rd), RV_REG_T0), ctx);
    }
    break;
  ...
}
```

(c) Final (extracted) JIT implementation in C.

Figure 9: Excerpt of the BPF JIT for RV32 for compiling the "`ADD64_REG DST, SRC`" instruction.

Using this approach, we found two JIT fragments better than our manual implementation for compiling `ADD64_REG` (Figure 9) and `SUB64_REG`. In each case, the synthesized fragment emitted four instructions, whereas our manual implementation emitted five. We adopted the synthesized fragments in the JIT.

7 Experience

Figure 10 shows the code size of the Jitterbug framework and the interpreters for verifying JITs, all written in Rosette. We wrote the interpreters in an idiomatic way [36: §3.3], adding instructions as needed. We borrowed part of the BPF and RV64 semantics from Serval [68], but rewrote the interpreters to support symbolic instructions (§5); we wrote the others from scratch. We developed the metatheory for JIT correctness and the bitvector axiomatization using 1,492 lines of Lean code.

Component (in Rosette)	Lines of code
Jitterbug framework	1,825
BPF interpreter	471
Arm32 interpreter	1,265
Arm64 interpreter	1,166
RISC-V interpreter (32- and 64-bit)	1,571
x86 interpreter (32- and 64-bit)	2,299

Figure 10: Line counts of Jitterbug's components.

	JIT impl.		Spec.	Per-opcode verification time			
	C	DSL		Min	Max	Mean	Median
RV32	1,964	1,420	336	16	401	73	55
RV64	1,862	1,225	284	4	7,542	116	24
Arm32	1,620	839	192	23	925	130	99
Arm64	1,025	653	163	4	110	26	23
x86-32	1,683	1,074	185	24	488	122	109
x86-64	1,382	644	182	5	170	33	27

Figure 11: Line counts and per-opcode verification time (in seconds) of the BPF JITs for six architectures.

The primary application of Jitterbug is a new BPF JIT for RV32, which we wrote in the DSL, proved against the stepwise specification, and extracted to a C implementation. To validate the generality of Jitterbug, we ported the existing BPF JITs for RV64, Arm32, Arm64, x86-32, and x86-64 in the Linux kernel to Jitterbug for verification. Each port was line-by-line transcription from C to the DSL (and Rosette), emitting the same instructions as the original JIT. These ports did not cover the support for legacy instruction sets (e.g., those lacking atomic instructions mentioned in §4.3), compiling `TAIL_CALL`, or optimizing register saving.

Figure 11 lists the line counts for each BPF JIT. The specification effort comprises writing assumptions and invariants for the implementation (§5). Since Jitterbug performs verification for each source instruction opcode individually, we measured the per-opcode verification time, using an Intel Core i7-7700K CPU at 4.5 GHz, with Boolector 3.2.1 as the SMT solver [69]. Verification time across the JITs depends on many factors (e.g., the JIT implementation or solver), though architectural differences are a contributing factor. For example, the most time-consuming case is verifying the JIT for RV64 with BPF's `LD64_IMM` (loading a 64-bit immediate), which emits 307 types of blocks of RISC-V instructions; the JIT for x86-64 emits six types for the same opcode.

Below we describe our experience using Jitterbug for the BPF JITs and a previously verified JIT for a stack machine [65].

The BPF JIT for RV32. We chose to implement a BPF JIT for RV32 because there was not one in the Linux kernel. The development took five iterations of code reviews.

The first two iterations occurred in June 2019. We sent an initial implementation to kernel developers to gather feedback and gauge interest. The implementation was written in Rosette and manually translated to C, and was unverified. The feedback was positive, with suggestions to add support for eliminating zero extensions [93], an optimization BPF had just introduced.

We submitted the third implementation in February 2020. It was switched to using the DSL (§6), which was less prone to errors in manual translation. It passed the BPF selftests suite, and was verified against an early version of the specification. One of the suggestions from kernel developers was to factor out the common code to be shared among the JITs, such as the per-instruction structure (§4.2). We addressed the suggestions in the next two iterations, after which the JIT was accepted into the Linux kernel (see §A.1.1).

Throughout this process, we refined both the specification and the implementation. The early version of the specification missed two bugs that were also missed by testing. The first bug was an off-by-one error for `TAIL_CALL`: the emitted instructions limited the `TCC` (tail-call counter) to 32, rather than the correct value 33. The second bug was that the JIT did not maintain 16-byte alignment of the stack as per the calling convention. We found the two bugs once we completed the specification.

Automated verification supported this development process in two ways. First, it minimized the proof burden for developing the JIT, which must be feature-complete for deployability. Second, it enabled us to catch up with new features being introduced (e.g., support for eliminating zero extensions) and address code reviews by kernel developers in a timely manner.

Code review. As listed in §A.1.2, we found 16 new bugs in the existing BPF JITs, wrote patches that fix these bugs, and verified the fixed code. In addition, we found two new bugs in the Arm64 instruction encoding library, a core component shared by BPF and other kernel subsystems (e.g., KVM). We wrote new test cases to be included in the BPF selftests suite. This is useful for catching similar bugs across the JITs, as various "bots" are running selftests continuously for the Linux kernel. Finding subtle bugs in well-tested code shows the effectiveness of the specification and verification.

The main effort for porting and verifying these JITs was in writing the target interpreters for Jitterbug. Verifying the JITs for Arm32 and Arm64 took one week each. Verifying the JITs for x86-32 and x86-64 took three weeks in total, due to the complexity of the x86 interpreter (e.g., instruction decoding). Translating C code to Rosette was mechanical and straightforward, though mistakes in manual translation might hide bugs; extending Jitterbug to work on C code is future work. For specification, we adopted the assumptions and invariants for the JIT for RV32 and adjusted them accordingly.

In our experience, automated verification is key to rapid code review using formal methods. As an example, in December 2019, the developer of the BPF JIT for RV64 submitted patches to add support for far jumps. We ported the patches to Jitterbug and verified their correctness within days of the patch submission. We reported the verification results to kernel developers; the patches were accepted with our review.

Optimization. Another advantage of verification is that it enables developing complex optimizations by providing a high degree of confidence in their correctness. As listed in §A.1.3, we developed 12 patches optimizing the existing BPF JITs. Like code review, we verified the correctness of these optimizations by manually translating the C code to Rosette.

One of the optimizations adds support for RISC-V compressed instruction-set extension (RVC) to the BPF JIT for RV64. RVC improves code density and reduces instruction cache misses by adding short 2-byte instructions for common operations [95: §5], but it poses a challenge to verification: the JIT may choose either base (4-byte) or RVC (2-byte) for emitting each instruction, depending on the immediate value or registers. This leads to an exponential increase in the number of paths in the JIT, emitted instructions, and machine state (e.g., variable code lengths causing the program counter to take different values). Developing and verifying this optimization took approximately 3 weeks, following the proof strategy described in §5 to scale verification.

Beyond BPF JITs. While Jitterbug focuses on the BPF JITs, we also applied it to a JIT for a stack machine to x86-32. We ported the "version 1" JIT described by Myreen [65] to the Jitterbug DSL and extracted it to C code; the port emitted the same x86-32 instructions and was able to run the example as in the paper (Jitterbug does not support the "version 2" JIT that emits self-modifying code). For specification, we excluded registers from the invariants, since the stack machine had no registers; and modeled memory as a map from addresses to values without using traces, since the stack machine had no shared memory (§4.3). For verification, we wrote an interpreter for the stack machine and reused the x86 interpreter provided by Jitterbug. This process took one day.

Jitterbug reported two bugs in the JIT implementation: the offsets for two conditional jump instructions are given as 5 in the original paper, but we concluded that the correct value should be 8. We fixed the offsets and verification succeeded. We believe that both are typos in the paper, as our (fixed) JIT is consistent with the paper's HOL4 code and proof.

8 Reflection and conclusion

Our work on Jitterbug was inspired by an earlier effort, started in 2015, to use the Coq theorem prover to develop a verified BPF JIT for x86-64. We chose to implement the JIT itself in x86-64 so as to minimize the trusted computing base. In hindsight, this was a mistake: doing so required reasoning about low-level machine state for both the compiler and emitted code, which hindered the completion of the proof; and the JIT implementation was impractical to audit and deploy due to the lack of C code and the optimizations seen in the Linux kernel. We suspended this effort two years later, in 2017.

Our interest in BPF JITs was revived with the development of symbolic profiling [10] and optimization [68, 74], which together demonstrated a systematic approach for scaling automated verification of low-level code. As an experiment, we wrote a bug finder for BPF JITs in Serval, which checked for strict equivalence of registers (§4.3) over straight-line instructions (§5). It enabled us to find 15 bugs regarding `ALU` instructions in two BPF JITs, although it was insufficient for verification or finding the bugs described in §3.2 due to the lack of a correctness specification and proof strategy (e.g., support for symbolic instructions).

For Jitterbug, we spent most of our effort devising a specification of JIT correctness that is general enough to cover a broad range of in-kernel JITs (e.g., without requiring padding emitted instructions), expressive enough to catch real bugs, and amenable to automated verification. We found the use of the Lean theorem prover valuable for navigating this trade-off, developing several iterations of the stepwise specification and a proof that implies the correctness of compiling entire programs. It also improved our confidence in the axiomatization of bitvector operations.

A key lesson from Jitterbug is that deciding what *not* to verify is as important as deciding what to verify. For instance, while ideally we would write and verify the BPF JIT for RV32 in C directly, the use of the DSL enabled us to fine-tune symbolic evaluation, which was critical for scaling verification. If we could not scale verification to JITs written in the DSL, verifying JITs written in C would surely be out of reach. Inspired by seL4 [78] and Ironclad [36], we bridged the resulting gap through validation, separately verifying that the instruction encoding functions in C emitted the same bytes as their original DSL code.

Through this paper, we presented our experience with specifying and verifying BPF JITs, a critical and rapidly evolving component in the Linux kernel. Our experience demonstrates, for the first time, the feasibility of extending automated verification to a restricted but practically important class of compilers. The source code of Jitterbug and the JITs is publicly available at `https://github.com/uw-unsat/jitterbug`.

Acknowledgments

We thank Anish Athalye, Tej Chajed, Gernot Heiser, Zhihao Jia, Noah Moroze, Jared Roesch, Zach Tatlock, Nickolai Zeldovich, the anonymous reviewers, and our shepherd, Bryan Parno, for feedback that improved this paper. We also thank Andrew Waterman and Claire Wolf for discussion on RISC-V, and H. Peter Anvin, Daniel Borkmann, Will Deacon, Brian Gerst, Song Liu, Andy Shevchenko, Alexei Starovoitov, Björn Töpel, Jiong Wang, Yanqing Wang, and Marc Zyngier for reviewing our patches to the Linux kernel. This work was supported by NSF awards CCF-1651225, CCF-1836724, and CNS-1844807, and by a gift from the VMware University Research Fund.

References

[1] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 97–111, Boston, MA, July 2018.

[2] Arm. *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*, March 2020.

[3] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[4] Paul Beame and Vincent Liew. Towards verifying nonlinear integer arithmetic. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)*, pages 238–258, Heidelberg, Germany, July 2017.

[5] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46, Anaheim, CA, April 2005.

[6] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain, CO, December 1995.

[7] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 121–134, Renton, WA, July 2019.

[8] Dion Blazakis. Interpreter exploitation: Pointer inference and JIT spraying. In *Black Hat DC*, Arlington, VA, February 2010.

[9] Daniel Borkmann. bpf, x86, arm64: Enable jit by default when not built as always-on. Commit `81c22041d9f1`, Linux kernel, December 2019.

[10] James Bornholt and Emina Torlak. Finding code that explodes under symbolic evaluation. In *Proceedings of the 2018 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Boston, MA, November 2018.

[11] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 775–788, St. Petersburg, FL, January 2016.

[12] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 15–28, Boston, MA, June–July 2004.

[13] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–322, Carlsbad, CA, October 2018.

[14] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the 10th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 46–160, Portland, OR, June 1989.

[15] Haogang Chen, Cody Cutler, Taesoo Kim, Yandong Mao, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, Singapore, July 2013. 6 pages.

[16] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of Amazon s2n. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, pages 430–446, Oxford, United Kingdom, July 2018.

[17] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 5 1976.

[18] Jonathan Corbet. BPF at Facebook (and beyond). `https://lwn.net/Articles/801871/`, October 2019.

[19] Jonathan Corbet. Concurrency management in BPF. `https://lwn.net/Articles/779120/`, February 2019.

[20] Palmer Dabbelt, Stefan O'Rear, Kito Cheng, Andrew Waterman, Michael Clark, Alex Bradbury, David Horner, Max Nordlund, Karsten Merker, and Sam Elliott. RISC-V ELF psABI specification. `https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md`, August 2020.

[21] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, September 2011.

[22] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In *Proceedings of the 25th International Conference on Automated Deduction (CADE)*, pages 378–388, Berlin, Germany, August 2015.

[23] Jake Edge. A trio of fuzzers. https://lwn.net/Articles/705937/, November 2016.

[24] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the 17th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–170, Philadephia, PA, May 1996.

[25] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the 1996 ACM SIGCOMM Conference*, pages 53–59, Stanford, CA, August 1996.

[26] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, CO, December 1995.

[27] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 73–90, San Francisco, CA, May 2019.

[28] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 1st ACM EuroSys Conference*, pages 177–190, Leuven, Belgium, April 2006.

[29] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, March 2018.

[30] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 287–305, Shanghai, China, October 2017.

[31] Matt Fleming. A thorough introduction to eBPF. https://lwn.net/Articles/740157/, December 2017.

[32] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing OS abstraction. In *Proceedings of the 14th ACM EuroSys Conference*, Dresden, Germany, March 2019.

[33] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing JIT compilers for in-kernel DSLs. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV)*, pages 564–586, Los Angeles, CA, July 2020.

[34] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1069–1084, Phoenix, AZ, June 2019.

[35] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. Addison Wesley, 2020.

[36] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, October 2014.

[37] Gernot Heiser, Gerwin Klein, and June Andronick. seL4 in Australia: From research to real-world trustworthy systems. *Communications of the ACM*, 63(4):72–75, April 2020.

[38] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 54–66, Heraklion, Greece, December 2018.

[39] Jingmei Hu, Eric Lu, David A Holland, Ming Kawaguchi, Stephen Chong, and Margo I. Seltzer. Trials and tribulations in synthesizing operating systems. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, pages 67–73, Huntsville, Ontario, Canada, October 2019.

[40] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the*

16th ACM Symposium on Operating Systems Principles (SOSP), pages 52–65, Saint-Malo, France, October 1997.

[41] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *Proceedings of the 9th European Congress Embedded Real-Time Software and Systems*, pages 1–9, Toulouse, France, January 2018.

[42] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, University of Washington, November 1991.

[43] Jakub Kicinski and Nicolaas Viljoen. eBPF hardware offload to SmartNICs: cls_bpf and XDP. In *the 3rd Technical Conference on Linux Networking*, Tokyo, Japan, October 2016.

[44] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[45] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.

[46] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–70, February 2014.

[47] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. Formally verified software in the real world. *Communications of the ACM*, 61(10):68–77, October 2018.

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 19–37, San Francisco, CA, May 2019.

[49] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory of Computing Systems*, 59(2):323–376, August 2016.

[50] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag, 2008.

[51] Butler W. Lampson and Robert F. Sproull. An open operating system for a single-user machine. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 98–105, Pacific Grove, CA, December 1979.

[52] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 216–226, Edinburgh, United Kingdom, June 2014.

[53] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[54] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, December 2009.

[55] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB computer safely and efficiently. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 234–251, Shanghai, China, October 2017.

[56] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent program. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210, London, United Kingdom, June 2020.

[57] Marek Majkowski. Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/, May 2019.

[58] Michaël Marcozzi, Qiyi Tang, Alastair Donaldson, and Cristian Cadar. Compiler fuzzing: How much does it matter? In *Proceedings of the 2019 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Athens, Greece, October 2019.

[59] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, Palo Alto, CA, October 1987.

[60] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, pages 191–201, Litchfield Park, AZ, December 1989.

[61] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 259–270, San Diego, CA, January 1993.

[62] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 39–51, Austin, TX, November 1987.

[63] J Strother Moore. Piton: A verified assembly level language. Technical Report 22, Computational Logic, Inc., September 1988.

[64] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 448–461, Santa Barbara, CA, June 2016.

[65] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, pages 107–118, Madrid, Spain, January 2011.

[66] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Extensible proof-producing compilation. In *Proceedings of the 18th International Conference on Compiler Construction*, pages 2–16, York, United Kingdom, March 2009.

[67] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, WA, October 1996.

[68] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 225–242, Huntsville, Ontario, Canada, October 2019.

[69] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 9:53–58, 2015.

[70] Jan Nordholz. Design of a symbolically executable embedded hypervisor. In *Proceedings of the 15th ACM EuroSys Conference*, Heraklion, Greece, April 2020.

[71] Justin Pettit, Ben Pfaff, Joe Stringer, Cheng-Chun Tu, Brenden Blanco, and Alex Tessmer. Bringing platform harmony to VMware NSX. *ACM SIGOPS Operating Systems Review*, 52(1):123–128, August 2018.

[72] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–310, Atlanta, GA, April 2016.

[73] Rob Pike, Bart N. Locanthi, and John Reiser. Hardware/-software trade-offs for bitmap graphics on the Blit. *Software: Practice and Experience*, 15(2):131–151, February 1985.

[74] Sorawee Porncharoenwase, James Bornholt, and Emina Torlak. Fixing code that explodes under symbolic evaluation. In *Proceedings of the 21st International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, New Orleans, LA, January 2020.

[75] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 983–1002, San Francisco, CA, May 2020.

[76] Dennis M. Ritchie. An incomplete history of the QED text editor. https://www.bell-labs.com/usr/dmr/www/qed.html, February 2004.

[77] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–227, Seattle, WA, October 1996.

[78] Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, Seattle, WA, June 2013.

[79] KP Singh. MAC and Audit policy using eBPF (KRSI). https://lkml.org/lkml/2020/3/28/479, March 2020.

[80] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Proceedings of the 21st International*

Conference on Theorem Proving in Higher Order Logics (TPHOLs), pages 28–32, Montreal, Canada, August 2008.

[81] Louis Sobel. eJitk: Extending Jitk to eBPF. https://css.csail.mit.edu/6.888/2015/papers/ejitk_sobel.pdf, May 2015.

[82] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, San Jose, CA, October 2006.

[83] Alexei Starovoitov. net: filter: rework/optimize internal bpf interpreter's instruction set. Commit bd4cf0ed331a, Linux kernel, March 2014.

[84] Alexei Starovoitov. bpf: introduce BPF_JIT_ALWAYS_ON config. Commit 290af86629b2, Linux kernel, January 2018.

[85] Chuck P. Thacker, Edward M. McCreight, Butler W. Lampson, Robert F. Sproull, and David R. Boggs. Alto: A personal computer. Technical Report CSL-79-11, Xerox Palo Alto Research Center, August 1979.

[86] The Coq Development Team. *The Coq Proof Assistant, version 8.12.0*, July 2020. URL https://doi.org/10.5281/zenodo.4021912.

[87] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[88] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Onward!*, pages 135–152, Boston, MA, October 2013.

[89] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, United Kingdom, June 2014.

[90] Lourival Vieira Neto, Roberto Ierusalimschy, Ana Lúcia de Moura, and Marc Balmer. Scriptable operating systems with Lua. In *Proceedings of the 10th Dynamic Languages Symposium*, pages 2–10, Portland, OR, October 2014.

[91] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, December 1993.

[92] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proceedings of the 1996 ACM SIGCOMM Conference*, pages 40–52, Stanford, CA, August 1996.

[93] Jiong Wang. bpf: eliminate zero extensions for sub-register writes. https://lore.kernel.org/bpf/1558736728-7229-1-git-send-email-jiong.wang@netronome.com/, May 2019.

[94] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–47, Broomfield, CO, October 2014.

[95] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, January 2016.

[96] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. RISC-V Foundation, December 2019.

[97] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, San Jose, CA, June 2011.

[98] William D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Computational Logic, Inc., October 1988.

[99] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

A Artifact appendix

A.1 Patches to the Linux kernel developed using Jitterbug

The following tables list the upstreamed patches to the Linux kernel that we have developed using Jitterbug.

A.1.1 Development of the BPF JIT for RV32

Commit	Architecture	Description
5f316b65e99f	RV32	Add RV32G eBPF JIT
ca6cb5447cec	RV32	Factor common RISC-V JIT code
745abfaa9eaf	RV32	Fix tail call count off by one in RV32 BPF JIT
91f658587a96	RV32	Fix stack layout of JITed code on RV32

A.1.2 Bug fixes and new test cases

Commit	Architecture	Description
bb9562cf5c67	Arm32	Fix bugs with ALU64 RSH, ARSH BPF_K shift by 0
4178417cc535	Arm32	Fix offset overflow for BPF_MEM BPF_DW
579d1b3faa37	Arm64	Fix two bugs in encoding 32-bit logical immediates
1e692f09e091	RV64	Clear high 32 bits for ALU32 add/sub/neg/lsh/rsh/arsh
489553dd13a8	RV64	Fix offset range checking for auipc+jalr on RV64
6fa632e719ee	x86-32	Fix bug with ALU64 LSH, RSH, ARSH BPF_K shift by 0
68a8357ec15b	x86-32	Fix bug with ALU64 LSH, RSH, ARSH BPF_X shift by 0
80f1f8503635	x86-32	Fix bug with JMP32 JSET BPF_X checking upper bits
5fa9a98fb103	x86-32	Fix incorrect encoding in BPF_LDX zero-extension
50fe7ebb6475	x86-32	Fix clobbering of dst for BPF_JSET
aee194b14dd2	x86-64	Fix encoding for lower 8-bit registers in BPF_STX BPF_B
d2b6c3ab70db	–	Add test for BPF_STX BPF_B storing R10
93e5fbb18cec	–	Add test for JMP32 JSET BPF_X with upper bits set
ac8786c72eba	–	Add tests for shifts by zero

A.1.3 Optimizations for existing BPF JITs

Commit	Architecture	Description
cf48db69bdfa	Arm32	Optimize ALU64 ARSH X using orrpl conditional instruction
c648c9c7429e	Arm32	Optimize ALU ARSH K using asr immediate instruction
fd49591cb49b	Arm64	Optimize AND,OR,XOR,JSET BPF_K using arm64 logical immediates
fd868f148189	Arm64	Optimize ADD,SUB,JMP BPF_K using arm64 add/sub immediates
46dd3d7d287b	RV64	Enable zext optimization for more RV64G ALU ops
0224b2acea0f	RV64	Enable missing verifier_zext optimizations on RV64
21a099abb765	RV64	Optimize FROM_LE using verifier_zext on RV64
ca349a6a104e	RV64	Optimize BPF_JMP BPF_K when imm == 0 on RV64
073ca6a0369e	RV64	Optimize BPF_JSET BPF_K using andi on RV64
bfabff3cb0fe	RV64	Modify JIT ctx to support compressed instructions
804ec72c68c8	RV64	Add encodings for compressed instructions
18a4d8c97b84	RV64	Use compressed instructions in the rv64 JIT

A.2 Bug-fixing commits in BPF JITs in the Linux kernel (May 2014–April 2020)

The following table lists bug-fixing commits in the BPF JITs in the Linux kernel for Arm32, Arm64, RV64, x86-32, and x86-64. The superscripts *J* and *S* mark those for fixing bugs found using Jitterbug and the BPF bug finder in Serval, respectively.

Commit	Architecture	Year	Description
ALU:			
bb9562cf5c67[J]	Arm32	2020	Fix bugs with alu64 rsh, arsh bpf_k shift by 0
14e589ff4aa3	Arm64	2015	Fix mod-by-zero case
251599e1d690	Arm64	2015	Fix div-by-zero case
d63903bbc30c	Arm64	2015	Fix endianness conversion bugs
1e4df6b72081	Arm64	2015	Fix signedness bug in loading 64-bit immediate
1e692f09e091[S]	RV64	2019	Clear high 32 bits for alu32 add/sub/neg/lsh/rsh/arsh
fe121ee531d1	RV64	2019	Clear target register high 32-bits for and/or/xor on alu32
6fa632e719ee[S]	x86-32	2019	Fix bug with alu64 lsh, rsh, arsh bpf_k shift by 0
68a8357ec15b[S]	x86-32	2019	Fix bug with alu64 lsh, rsh, arsh bpf_x shift by 0
b9aa0b35d878	x86-32	2019	Fix bug for bpf_alu64 | bpf_neg
343f845b3759	x86-64	2015	Fix from_be16 and from_le16/32 instructions
JMP:			
2b589a7e2bd3	Arm32	2018	Correct check_imm24
ddc665a4bb4b	Arm64	2017	Fix jit branch offset related to ldimm64
8eee539ddea0	Arm64	2015	Fix out-of-bounds read in bpf2a64_offset()
50fe7ebb6475[J]	x86-32	2020	Fix clobbering of dst for bpf_jset
80f1f8503635[J]	x86-32	2020	Fix bug with jmp32 jset bpf_x checking upper bits
711aef1bbf88	x86-32	2019	Fix bug for bpf_jmp | bpf_jsgt, bpf_jsle, bpf_jslt, bpf_jsge
7c2e988f400e	x86-64	2019	Fix x64 jit code generation for jmp to 1st insn
MEM:			
4178417cc535[J]	Arm32	2020	Fix offset overflow for bpf_mem bpf_dw
ec19e02b343d	Arm32	2018	Fix ldx instructions
8968c67a82ab	Arm64	2019	Remove prefetch insn in xadd mapping
7005cade1bdb	Arm64	2017	Use separate register for state in stxr
5ca1ca01fae1	x86-32	2020	Fix logic error in bpf_ldx zero-extension
5fa9a98fb103[J]	x86-32	2020	Fix incorrect encoding in bpf_ldx zero-extension
aee194b14dd2[J]	x86-64	2020	Fix encoding for lower 8-bit registers in bpf_stx bpf_b
CALL:			
8c11ea5ce13d	Arm64	2018	Fix getting subprog addr from aux for calls
489553dd13a8[J]	RV64	2020	Fix offset range checking for auipc+jalr on rv64
TAIL_CALL and EXIT:			
02088d9b392f	Arm32	2018	Fix register saving
f4483f2cc1fd	Arm32	2018	Fix tail call jumps
51c9fbb1b146	Arm64	2014	Lift restriction on last instruction
16338a9b3ac3	Arm64	2018	Fix out of bounds access in tail call
a2284d912bfc	Arm64	2018	Fix stack_depth tracking in combination with tail calls
d8b54110ee94	Arm64	2017	Fix faulty emission of map access in tail calls
96bc4432f5ad	RV64	2019	Limit to 33 tail calls
769e0de6475e	x86-64	2014	Fix epilogue generation for ebpf programs
90caccdd8cc0	x86-64	2017	Fix bpf_tail_call() x64 jit
2482abb93ebf	x86-64	2015	Fix general protection fault when tail call is invoked
Prologue and epilogue:			
d1220efd2348	Arm32	2018	Fix stack alignment
f1003b787c00	RV64	2019	Fix broken bpf tail calls
9e4e5b5c8666	x86-32	2018	Fix regression caused by commit 24dea04767e6
fe8d9571dc50	x86-64	2019	Fix stack layout of jited bpf code





COBRA: Making Transactional Key-Value Stores Verifiably Serializable

Cheng Tan, Changgeng Zhao, Shuai Mu*, and Michael Walfish

NYU Department of Computer Science, Courant Institute *Stony Brook University

Abstract. Today's cloud databases offer strong properties, including serializability, sometimes called the gold standard database correctness property. But cloud databases are complicated black boxes, running in a different administrative domain from their clients. Thus, clients might like to know whether the databases are meeting their contract. To that end, we introduce COBRA; COBRA applies to transactional key-value stores. It is the first system that combines (a) black-box checking, of (b) serializability, while (c) scaling to real-world online transactional processing workloads. The core technical challenge is that the underlying search problem is computationally expensive. COBRA tames that problem by starting with a suitable SMT solver. COBRA then introduces several new techniques, including a new encoding of the validity condition; hardware acceleration to prune inputs to the solver; and a transaction segmentation mechanism that enables scaling and garbage collection. COBRA imposes modest overhead on clients, improves over baselines by 10× in verification cost, and (unlike the baselines) supports *continuous* verification. Our artifact can handle 2000 transactions/sec, equivalent to 170M/day.

1 Introduction and motivation

A new class of cloud databases has emerged, including Amazon DynamoDB and Aurora [2, 4, 133], Azure CosmosDB [7], CockroachDB [9], YugaByte DB [36], and others [16, 17, 21, 22, 69]. Compared to earlier generations of NoSQL databases (such as Facebook Cassandra, Google Bigtable, and Amazon S3), members of the new class offer the same scalability, availability, replication, and geo-distribution but in addition offer *serializable* transactions [55, 110]: all transactions appear to execute in a single, sequential order.

Serializability is the gold-standard isolation level [48, 77], and the correctness contract that many applications and programmers implicitly assume: their code would be incorrect if the database provided a weaker contract [137]. Note that serializability encompasses weaker notions of correctness, like basic integrity: if a returned value does not read from a valid write, that will manifest as a non-serializable result. Serializability also implies that the database handles failures robustly: non-tolerated server failures, particularly in the case of a distributed database, are a potential source of non-serializable results.

However, a user of a cloud database can legitimately wonder whether the database in fact provides the promised contract. For one thing, users often have no visibility into a cloud database's implementation. In fact, even when the source code is available [9, 16, 17, 36], that does not necessarily yield visibility: if the database is hosted by someone else, you can't really be sure of its operation. Meanwhile, any internal corruption—as could happen from misconfiguration, operational error, compromise, or adversarial control at any layer of the execution stack—can cause a serializability violation. Beyond that, one need not adopt a paranoid stance ("the cloud as malicious adversary") to acknowledge that it is difficult, as a technical matter, to provide serializability *and* geo-distribution *and* geo-replication *and* high performance under various failures [40, 78, 147]. Doing so usually involves a consensus protocol that interacts with an atomic commit protocol [69, 96, 103]—a complex combination, and hence potentially bug-prone. Indeed, today's production systems have exhibited serializability violations [1, 18, 19, 25, 26] (see also §6.1).

This leads to our core question: *how can clients verify the serializability of a black-box database?* To be clear, related questions have been addressed before. The novelty in our problem is in combining three aspects:

(a) Black box, unmodified database. In our setting, the database does not "know" it's being checked; the input to the verification machinery will be only the inputs to, and outputs from, the database. This matches the cloud context (even when the database is open source, as noted above), and contrasts with work that checks for isolation or consistency anomalies by using "inside information" [62, 86, 109, 123, 130, 141, 143], for example, access to internal scheduling choices. Also, we target production workloads and standard key-value APIs (§2).

(b) Serializability. We focus on serializability, in contrast to weaker isolation levels. Serializability has a strict variant and a non-strict variant [56, 110]; in the former, the effective transaction order must be consistent with real time. We attend to both variants in this paper. However, the weight is on the non-strict variant, as it poses a more difficult computational problem; the strict variant is "easier" because the real-time constraint diminishes the space of potentially-valid execution schedules.

On the one hand, the majority of databases that offer serializability offer the strict variant. On the other hand, checking non-strict serializability is germane, for two reasons. First, some databases claim to provide the non-strict variant (in general [11], or under clock skew [35], or for read-only workloads [32]), while others don't specify the variant [3, 5]. Second, the strict case can degenerate to the non-strict case. Heavy concurrency, for example, means few real-time constraints, so the difficult computational problem re-enters. As a special case, clock drift causes otherwise ordered transactions to be concurrent (§3.5, §6.1).

(c) Scalability. This means, first, scaling to real-world online transactional processing workloads at reasonable cost. It also

means incorporating mechanisms that enable a verifier to work incrementally and to keep up with an ever-growing history.

However, aspects (a) and (b) set up a challenge: checking black-box serializability has long been known to be NP-complete [54, 110]. Recent work of Biswas and Enea (BE) [59] lowered the complexity to polynomial time, under natural restrictions (which hold in our context); see also pioneering work by Sinha et al. [124] (§7). However, these two approaches don't meet our goal of scalability. For example, in BE, the number of clients appears in the exponent of the algorithm's running time (§6, §7) (e.g., 14 clients means the algorithm is $O(n^{14})$). Furthermore, even if there were a small number of clients, BE does not include mechanisms for handling a continuous and ever-growing history.

Despite the computational complexity, there is cause for hope: one of the remarkable developments in the field of formal verification has been the use of heuristics to "solve" problems whose general form is intractable. This owes to major advances in solvers (advanced SAT and SMT solvers) [49, 57, 64, 73, 84, 99, 107, 128], coupled with an explosion of computing power. Thus, our guiding intuition is that it ought to be possible to verify serializability in many real-world cases. This paper describes a system called COBRA, which starts from this intuition, and provides a solution to the problem posed by (a)–(c).

COBRA applies to transactional key-value stores (everywhere in this paper it says "database", this is what we mean). COBRA consists of a third-party, unmodified database that is not assumed to "cooperate"; a set of legacy database clients that COBRA modifies to link to a library; one or more *history collectors* that are assumed to record the actual requests to and responses from the database; and a *verifier* that comprehensively checks serializability, in a way that "keeps up" with the database's (average) load. The database is untrusted while the clients, collectors, and verifier are all in the same trust domain (for example, deployed by the same organization). Section 2 further details the setup and gives example scenarios. COBRA solves two main problems:

1. Efficient witness search (§3). A brute-force way to validate serializability is to demonstrate the existence of a graph G whose nodes are transactions in the history and whose edges meet certain *constraints*, one of which is acyclicity (§2.3). From our starting intuition and the structure of the constraints, we are motivated to use a SAT or SMT solver [34, 50, 73, 127]. But complications arise. To begin with, encoding acyclicity in a SAT instance brings overhead [79, 80, 91] (we see this too; §6.1). Instead, COBRA uses a recent SMT solver, MonoSAT [52], that is well-suited to checking graph properties (§3.4). However, using MonoSAT alone on the aforementioned brute-force search problem is still too expensive (§6.1).

To address this issue, COBRA develops domain-specific pruning techniques and reduces the search problem size. First, COBRA introduces a new encoding that exploits common patterns in real workloads, such as read-modify-write transactions, to efficiently infer ordering relationships from a history (§3.1–§3.2). (We prove that COBRA's encoding is a valid reduction in Appendix B [132].) Second, COBRA uses parallel hardware (our implementation uses GPUs; §5) to compute *all-pairs reachability* over a graph whose nodes are transactions and whose edges are known precedence relationships; then, COBRA resolves some of the constraints efficiently, by testing whether a candidate edge would generate a cycle with an existing path.

2. Scaling to a continuous and ever-growing history (§4). Online cloud databases run in a continuous fashion, where the corresponding history is uninterrupted and grows unboundedly. To support online databases, COBRA verifies in rounds. From round-to-round, the verifier checks serializability on a portion of the history. However, the challenge is that the verifier seemingly needs to involve all history, because serializability does not respect real-time ordering, so future transactions can read from values that (in a real-time view) have been overwritten. To solve this problem, clients issue periodic *fence transactions* (§4.2). The epochs impose coarse-grained synchronization, creating a window from which future reads, if they are to be serializable, are permitted to read. This allows the verifier to discard transactions prior to the window.

We implement COBRA (§5) and experiment with it on production databases with various workloads (§6). COBRA detects all serializability violations we collect from real systems' bug reports. COBRA's core (single-round) verification improves on baselines by 10× in the problem size it can handle for a given time budget. For example, COBRA finishes checking 10k transactions in 14 seconds, whereas baselines can handle only 1k or less in the same time budget. For an online database with continuous traffic, COBRA achieves a sustainable verification throughput of 2k txn/sec on the workloads that we experiment with (this corresponds to a workload of 170M/day; for comparison, Apple Pay handles 33M txn/day [6], and Visa handles 150M txn/day [33], admittedly for a slightly different notion of "transaction"). COBRA imposes modest overhead.

COBRA has several limitations (§8). First, there is no guarantee that COBRA terminates in reasonable time (though our experiments on real workloads do). Second, COBRA supports only a key-value API, and thus does not handle range queries and SQL operations such as "join" and "sum" (though one can translate these queries and operations to a key-value API, as commonly done in research on transactional key-value stores [100, 108, 129, 136, 140, 144]). Third, COBRA does not yet support async (event-driven) I/O patterns in clients (only multithreading). Fourth, COBRA mostly punts fault-tolerance of the verifier and collectors (though modular solutions exist). Finally, we have not identified serializability violations in the wild. Of course, that does not mean that databases unfailingly execute correctly. Indeed, COBRA gives us a way, for the first time, to get confidence in the observed executions of these black box databases.

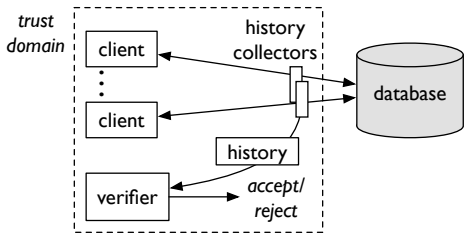


Figure 1: Cobra's architecture. The dashed rectangle is a trust domain. The verifier is off the critical path but must keep up on average.

2 Overview and technical background

2.1 Setup and scenarios

Figure 1 depicts cobra's architecture. *Clients* issue requests to a *database* (a transactional key-value store) and receive results. The database is untrusted: the results can be arbitrary.

Each client request is one of five operations: start, commit, abort (which refer to *transactions*), and read and write (which refer to *keys*).

History collectors sit between clients and the database, capturing the requests that clients issue and the (possibly wrong) results delivered by the database. This capture is a *fragment of a history*. A *history* is a set of operations; it is the union of the fragments from all collectors.

A *verifier* retrieves history fragments from collectors and attempts to verify whether the history is *serializable*; we make this term precise below (§2.2).

The verifier proceeds in *rounds*; each round consists of a witness search, the input to which is logically the output of the previous round and new history fragments. The verifier must work against an online and continuously available database; however, the verifier performs its work in the background, off the critical path.

The verifier requires the full history including all requests to, and responses from, the database. Cobra assumes that neither the verifier nor the collectors crash (we revisit in §8).

Clients issue operations to a database through *sessions*; a client can have multiple simultaneous sessions. Within a session, transactions do not overlap (requests are blocking). Thus, a client can be multithreaded but not event-driven.

Clients, history collectors, and the verifier are in the same trust domain. This architecture is relevant in real-world scenarios. Consider for example an enterprise web application whose end-users are geo-distributed employees of the enterprise. The application servers run on the enterprise's hardware while the back-end of the web application is a cloud database [27]. Note that our *clients* are the application servers, as clients of the database. A similarly structured example is online gaming, where the main program runs on company-owned servers while the user data is stored in a cloud database [24].

In these scenarios, the verifier runs on hardware belonging to the trust domain. There are several options, meanwhile, for the collectors. Collectors can be middleboxes situated at the edge of the enterprise or gaming company, allowing them to capture the requests and responses between the database clients and the cloud database. Another option is to run the collector in an untrusted cloud, using a Trusted Execution Environment (TEE), such as Intel's SGX. Recent work has demonstrated such a collector [46], as a TLS proxy that logs inbound and outbound messages, thereby ensuring (via the fact that clients expect the server to present a valid SSL/TLS certificate) that clients' requests and responses are indeed logged.

Verifier's performance requirement. The verifier's performance will be reported as capacity (transactions/sec); this capacity must be at least the average offered load seen by the database over some long interval, for example a day. Note that the verifier's capacity need not match the database's *peak* load: because the verifier is off the critical path, it can catch up.

2.2 Verification problem statement

Preliminaries. We work within Adya's canonical framework for specifying isolation levels [38], as summarized below.

First, assume that each database write creates a unique *version* for the given key, and each transaction reads and writes a key at most once; thus, any read can be associated with the transaction that issued the corresponding write. Cobra discharges this assumption in the client library (§5), which embeds a unique id in each write and consumes the id on a read.

In Adya's formalism, a *history* is a set of operations performed by transactions (as in cobra, §2.1), together with a *version order* [38, §3.1.2]: for each key, a total order of committed versions. The version order comes from within the database and is not exposed externally. In cobra, history is collected outside the database so doesn't contain a version order. (Cobra's history is also known as a *multi-version log* [54], as discussed in Appendix B [132]).

A history is *serializable*, if there exists a total order on the committed transactions such that executing transactions in this order produces the same result as in the history (in Adya's formalism, an additional requirement for serializability is that the aforementioned total order is consistent with the given version order). *Strictly serializable* [110] is the same as serializable, except that the total order must also obey real time: if a transaction T_i commits before T_j starts in real time, T_i appears earlier than T_j in the total order.

A history imposes *dependencies*. Specifically, a history (without a version order) induces *read-dependencies* (a transaction T_j reads the value written by transaction T_i, denoted $T_i \rightarrow T_j$). Adding a version order yields two other kinds of dependencies: *write-dependency* (T_i writes a key, and T_j overwrites it, so $T_i \rightarrow T_j$), and *anti-dependency* (T_i reads a value that is overwritten by T_j, so $T_i \rightarrow T_j$).

A *serialization graph* (of a history and a given version order) is a graph whose vertices are all transactions in the history and edges are all dependencies described above. Note that aborted and ongoing transactions are not in the serialization graph.

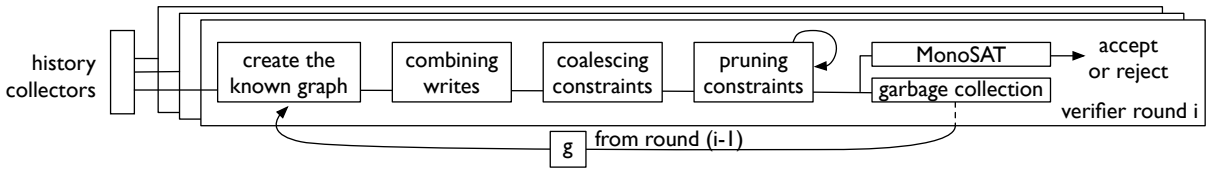


Figure 2: The verifier's process, within a round and across rounds.

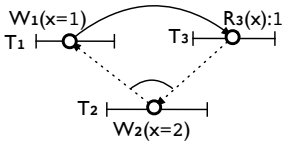
The core problem. An important fact is that a history H is serializable iff there exists a version order such that the serialization graph arising from H and that version order is acyclic [54]. Based on this fact, the core problem is to identify such a serialization graph (that arises from H and some version order), or assert that none exists.

Notice that this problem would be straightforward if the database revealed its internal ordering, thus deciding a version order and all dependencies: one could construct the corresponding serialization graph, and test it for acyclicity. Indeed, that is a well-established family of techniques [65, 138]. But the version order is unavailable in our context, so we have to consider all possible version orders, and test whether any of the implied sets of dependencies yields an acyclic serialization graph.

2.3 Starting point: intuition and brute force

This section describes a brute-force solution, which serves as the starting point for COBRA and gives intuition. The approach relies on a data structure called a *polygraph* [110], which captures unknown dependencies.

In a polygraph, vertices (V) are transactions and edges (E) are read-dependencies. Note that read-dependencies are evident from the history because values are unique, by assumption (§2.2). There is a set, C, which we call *constraints*, that captures possible (but unknown) dependencies. Here is an example polygraph:



It has three vertices $V = \{T_1, T_2, T_3\}$, one known edge in $E = \{(T_1, T_3)\}$ from the known read-dependency $W_1(x) \longrightarrow R_3(x)$, and one constraint $\langle (T_3, T_2),\ (T_2, T_1) \rangle$ which is shown as two dashed arrows connected by an arc. This constraint captures the fact that T_2 cannot happen in between T_1 and T_3, because otherwise T_3 should have read x from T_2 instead of from T_1. Hence T_2 has to happen either after T_3 or before T_1, but it is unknown which option is the truth.

Formally, a *polygraph* $P = (V,\ E,\ C)$ is a directed graph $(V,\ E)$ which we call the *known graph*, together with a set of *bipaths*, C; that is, pairs of edges—not necessarily in E—of the form $\langle (v, u),\ (u, w) \rangle$ such that $(w, v) \in E$. A bipath of that form can be read as "either u happened after v, or else u happened before w". Now, define *the polygraph* (V, E, C) *associated with a history*, as follows [138]:

- V are all committed transactions in the history
- $E = \{(T_i, T_j) \mid T_j \text{ reads from } T_i\}$. Notation: $T_i \xrightarrow{\text{wr(x)}} T_j$, for some x.
- $C = \{\langle (T_j, T_k),\ (T_k, T_i) \rangle \mid (T_i \xrightarrow{\text{wr(x)}} T_j) \wedge (T_k \text{ writes to } x) \wedge T_k \neq T_i \wedge T_k \neq T_j\}$.

The edges in E capture all read-dependencies, which as noted are evident from the history. C captures how uncertainty is encoded into constraints. Specifically, for each read-dependency in the history, all other transactions that write the same key happen either after the given read or before the given write.

A directed graph is called *compatible* with a polygraph if the graph has the same nodes and known edges in the polygraph, and the graph chooses one edge out of each constraint; one can think of such a graph as a solution to the constraint satisfaction problem posed by the polygraph. Formally, a graph (V', E') is *compatible* with a polygraph (V, E, C) if: $V = V'$, $E \subseteq E'$, and $\forall \langle e_1, e_2 \rangle \in C,\ (e_1 \in E' \wedge e_2 \notin E') \vee (e_1 \notin E' \wedge e_2 \in E')$.

A crucial fact (proved in Appendix B [132]) is: there exists an acyclic directed graph that is compatible with the polygraph associated to a history H, iff there exists an acyclic serialization graph G of H. Furthermore, we have seen that if there is such an acyclic serialization graph for H, then H is serializable (§2.2). Putting these facts together yields a brute-force approach for verifying serializability: first, construct a polygraph from a history; second, search for a compatible graph that is acyclic. However, not only does this approach need to consider $|C|$ binary choices ($2^{|C|}$ possibilities) but also $|C|$ is massive: it is a sum of quadratic terms, specifically $\sum_{k \in \mathcal{K}} r_k \cdot (w_k - 1)$, where $\mathcal{K}$ is the set of keys in the history, and each r_k and w_k are the number of reads and writes of key k.

3 Verifying serializability in COBRA

Figure 2 depicts the major components of verification. This section covers one round of verification. As a simplification, assume that the round runs in a vacuum; Section 4 discusses how rounds are linked.

COBRA uses the MonoSAT SMT solver [52], which is geared to graph properties (§3.4). Nevertheless, the brute-force encoding (§2.3) would overwhelm even MonoSAT (§6.1).

COBRA refines that encoding in several ways. It introduces *write combining* (§3.1) and *coalescing* (§3.2). These techniques are motivated by common patterns that impose restrictions on the search space. COBRA's verifier also does its own inference (§3.3) before invoking the solver. This is motivated by observing that (a) all-pairs reachability information (in the "known edges") yields quick resolution of many constraints,

```
1:  procedure ConstructEncoding(history)
2:      g, readfrom, wwpairs ← CreateKnownGraph(history)
3:      con ← GenConstraints(g, readfrom, wwpairs)
4:      con, g ← Prune(con, g) // §3.3, executed one or more times
5:      return con, g
6:
7:  procedure CreateKnownGraph(history)
8:      g ← empty Graph                          // the known graph
9:      wwpairs ← Map {⟨Key, Tx⟩ → Tx}           // consecutive writes
10:     readfrom ← Map {⟨Key, Tx⟩ → Set⟨Tx⟩} // maps a write to its readers
11:     for transaction tx in history:
12:         g.Nodes += tx
13:         for read operation rop in tx:
14:             g.Edges += (rop.read_from_tx, tx)     // read-dependencies
15:             readfrom[⟨rop.key, rop.read_from_tx⟩] += tx
16:
17:         // detect RMW (read-modify-write) transactions
18:         for all Keys key that are both read and written by tx:
19:             rop ← the operation in tx that reads key
20:             if wwpairs[⟨key, rop.read_from_tx⟩] ≠ null:
21:                 REJECT     // multiple consecutive writes, not serializable
22:             wwpairs[⟨key, rop.read_from_tx⟩] ← tx
23:
24:     add session order edges to g     // §4.2
25:     return g, readfrom, wwpairs
26:
27: procedure GenConstraints(g, readfrom, wwpairs)
28:     // each key maps to set of chains; each chain is an ordered list
29:     chains ← empty Map {Key → Set⟨List⟩}
30:     for transaction tx in g:
31:         for write wrop in tx:
32:             chains[wrop.key] += [ tx ]     // one-element list
33:
34:     CombineWrites(chains, wwpairs)     // §3.1
35:     InferRWEdges(chains, readfrom, g) // infer anti-dependency
36:
37:     con ← empty Set
38:     for ⟨key, chainset⟩ in chains:
39:         for every pair {chain_i, chain_j} in chainset:
40:             con += Coalesce(chain_i, chain_j, key, readfrom) // §3.2
41:
42:     return con

43: procedure CombineWrites(chains, wwpairs)
44:     for ⟨key, tx_1, tx_2⟩ in wwpairs:
45:         // By construction of wwpairs, tx_1 is the write immediately
46:         // preceding tx_2 on key. Thus, we can sequence all writes
47:         // prior to tx_1 before all writes after tx_2, as follows:
48:         chain_1 ← the list in chains[key] whose last elem is tx_1
49:         chain_2 ← the list in chains[key] whose first elem is tx_2
50:         chains[key] \= {chain_1, chain_2}
51:         chains[key] += concat(chain_1, chain_2)
52:
53: procedure InferRWEdges(chains, readfrom, g)
54:     for ⟨key, chainset⟩ in chains:
55:         for chain in chainset:
56:             for i in [0, length(chain) − 2]:
57:                 for rtx in readfrom[⟨key, chain[i]⟩]:
58:                     if (rtx ≠ chain[i+1]): g.Edges += (rtx, chain[i+1])
59:
60: procedure Coalesce(chain_1, chain_2, key, readfrom)
61:     edge_set_1 ← GenChainToChainEdges(chain_1, chain_2, key, readfrom)
62:     edge_set_2 ← GenChainToChainEdges(chain_2, chain_1, key, readfrom)
63:     return ⟨edge_set_1, edge_set_2⟩
64:
65: procedure GenChainToChainEdges(chain_i, chain_j, key, readfrom)
66:     if readfrom[⟨key, chain_i.tail⟩] = ∅:
67:         edge_set ← {(chain_i.tail, chain_j.head)}
68:         return edge_set
69:
70:     edge_set ← empty Set
71:     for rtx in readfrom[⟨key, chain_i.tail⟩]:
72:         edge_set += (rtx, chain_j.head)
73:     return edge_set
74:
75: procedure Prune(con, g)
76:     // tr is the transitive closure (reachability of every two nodes) of g
77:     tr ← TransitiveClosure(g) // standard algorithm; see [70, Ch.25]
78:     for c =⟨edge_set_1, edge_set_2⟩ in con:
79:         if ∃(tx_i, tx_j) ∈ edge_set_1 s.t. tx_j ⇝ tx_i in tr:
80:             g.Edges ← g.Edges ∪ edge_set_2
81:             con −= c
82:         else if ∃(tx_i, tx_j) ∈ edge_set_2 s.t. tx_j ⇝ tx_i in tr:
83:             g.Edges ← g.Edges ∪ edge_set_1
84:             con −= c
85:     return con, g
```

Figure 3: Cobra's procedure for converting a history into a constraint satisfaction problem (§3). After this procedure, cobra feeds the results (a graph of known edges G and set of constraints C) to a constraint solver (§3.4), which searches for a graph that includes the known edges from G, meets the constraints in C, and is acyclic. We prove the algorithm's validity in Appendix B [132].

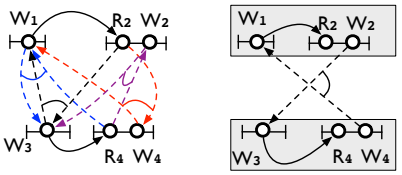
and (b) computing that information is amenable to acceleration on parallel hardware such as GPUs (§5).

Figure 3 depicts the algorithm that constructs cobra's encoding and shows how the techniques combine. Note that cobra relies on a generalized notion of constraints. Whereas previously a constraint was a pair of edges, now a constraint is a pair of *sets of edges*. Meeting a constraint $\langle A, B\rangle$ means including all edges in A and excluding all in B, or vice versa. More formally, we say that a graph (V', E') is *compatible* with a known graph $G = (V, E)$ and generalized constraints C if: $V = V'$, $E \subseteq E'$, and $\forall \langle A, B\rangle \in C, (A \subseteq E' \wedge B \cap E' = \emptyset) \vee (A \cap E' = \emptyset \wedge B \subseteq E')$.

We prove the validity of cobra's encoding in Appx B [132]. Specifically we prove that *there exists an acyclic graph that is compatible with the constraints constructed by* cobra *on a given history if and only if the history is serializable*.

3.1 Combining writes

Cobra exploits the read-modify-write (RMW) pattern, in which a transaction reads a key and then writes the same key. The pattern is common in real-world scenarios, for example shopping: in one transaction, get the number of an item in stock, decrement, and write back the number. Cobra uses RMWs to impose order on writes; this reduces the orderings that the verification procedure would otherwise have to consider. Here is an example:



There are four transactions, all operating on the same key. Two of the transactions are RMW, namely R_2, W_2 and R_4, W_4. On the left is the basic polygraph (§2.3). It has four constraints (each in a different color), derived from the two read-dependencies.

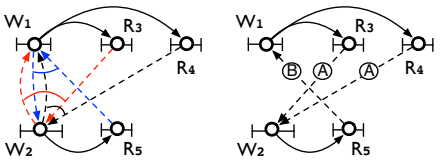
COBRA goes further, inferring *chains*. A single chain comprises a *sequence of transactions whose write operations are consecutive*; in the figure, a chain is indicated by a shaded area. Notice that the only ordering possibilities exist at the granularity of chains (rather than individual writes); in the example, the two possibilities of course are $[W_1, W_2] \rightarrow [W_3, W_4]$ and $[W_3, W_4] \rightarrow [W_1, W_2]$. This is a reduction in the possibility space; for instance, the original version considers the possibility that W_3 is immediately prior to W_1 (the upward dashed black arrow), but COBRA "recognizes" the impossibility of that.

To construct chains, COBRA initializes every write as a one-element chain (Figure 3, line 32). Then, COBRA consolidates chains: for each RMW transaction t and the transaction t' that contains the prior write, COBRA concatenates the chain containing t' and the chain containing t (lines 22 and 44–51).

If a transaction t, which is *not* an RMW, reads from a transaction u, then t requires an anti-dependency edge to u's successor (call it v); otherwise, t could appear in the graph downstream of v, which would mean t actually read from v (or even from a later write), which does not respect history. COBRA creates the needed edge $t \rightarrow v$ in InferRWEdges (Figure 3, line 53). Note that in the brute-force approach (§2.3), analogous edges appear as the first component in a constraint.

3.2 Coalescing constraints

This technique exploits the fact that, in many real-world workloads, there are far more reads than writes. At a high level, COBRA combines all reads that read-from the same write. We give an example and then generalize.



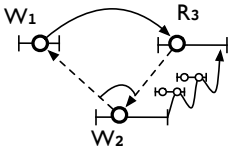
In the above figure, there are five single-operation transactions, to the same key. On the left is the basic polygraph (§2.3), which contains three constraints; each is in a different color. Notice that all three constraints involve the question: which write happened first, W_1 or W_2?

One can represent the possibilities as a constraint $\langle A', B' \rangle$ where $A' = \{(W_1, W_2), (R_3, W_2), (R_4, W_2)\}$ and $B' = \{(W_2, W_1), (R_5, W_1)\}$. In fact, COBRA does not include (W_1, W_2) because there is a known edge (W_1, R_3), which, together with (R_3, W_2) in A', implies the ordering $W_1 \rightarrow R_3 \rightarrow W_2$, so there is no need to include (W_1, W_2). Likewise, COBRA does not include (W_2, W_1) on the basis of the known edge (W_2, R_5). So COBRA includes the constraint $\langle A, B \rangle = \langle \{(R_3, W_2), (R_4, W_2)\}, \{(R_5, W_1)\} \rangle$ in the figure.

To construct constraints using the above reductions, COBRA does the following. Whereas the brute-force approach uses all reads and their prior writes (§2.3), COBRA considers particular *pairs of writes*, and creates constraints from these writes and their following reads. The particular pairs of writes are the first and last writes from all pairs of chains pertaining to that key. In more detail, given two chains, $chain_i, chain_j$, COBRA constructs a constraint c by (i) creating a set of edges ES_1 that point from reads of $chain_i$.tail to $chain_j$.head (Figure 3, lines 71–72); this is why COBRA does not include the (W_1, W_2) edge above. If there are no such reads, ES_1 is $chain_i$.tail $\rightarrow$ $chain_j$.head (Figure 3, line 67); (ii) building another edge set ES_2 that is the other way around (reads of $chain_j$.tail point to $chain_i$.head, etc.), and (iii) setting c to be $\langle ES_1, ES_2 \rangle$ (Figure 3, line 63).

3.3 Pruning constraints

Our final technique mines information that is encoded in paths in the known graph, to cull irrelevant possibilities en masse. The underlying logic is almost trivial. The interesting aspect is that the technique is enabled by a design decision to accelerate the computation of reachability on parallel hardware (§5 and Figure 3, line 77); this can be done since the computation is iterated (Boolean) matrix multiplication. Here is an example:



The constraint is $\langle (R_3, W_2), (W_2, W_1) \rangle$. Having precomputed reachability, COBRA knows that the first choice cannot hold, as it creates a cycle with the path $W_2 \rightsquigarrow R_3$; COBRA thereby concludes that the second choice holds. Generalizing, if COBRA determines that an edge in a constraint generates a cycle, COBRA throws away both components of the entire constraint and adds all the other edges to the known graph (Figure 3, lines 78–84). In fact, COBRA prunes multiple times, if necessary (§5).

3.4 Solving

The remaining step is to search for an acyclic graph that is compatible with the known graph and constraints, as computed in Figure 3. COBRA does this with a constraint solver. However, traditional solvers do not perform well on this task because encoding graph acyclicity as a set of SAT formulas is expensive (a claim by Janota et al. [91], which we also observed; §6.1).

COBRA instead uses MonoSAT, which is a particular kind of SMT solver [57] that includes SAT modulo *monotonic* theories [52]. This solver efficiently encodes and checks graph properties, such as acyclicity.

COBRA represents a verification problem instance (a graph G and constraints C) as follows. COBRA creates a Boolean variable $E_{(i,j)}$ for each vertex-vertex pair; True (resp., False) means the searched-for compatible graph has (resp., does not have) the given edge. For each edge in the known graph G, COBRA sets the corresponding Boolean variable to be True. For the

constraints C, recall that each constraint $\langle A, B \rangle$ is a pair of sets of edges, and represents a mutually exclusive choice to include either all edges in A or else all edges in B. Cobra encodes this in the natural way: $((\forall e_a \in A, e_a) \wedge (\forall e_b \in B, \neg e_b)) \vee ((\forall e_a \in A, \neg e_a) \wedge (\forall e_b \in B, e_b))$. (By abuse of notation, we have used e_a and e_b to refer to the corresponding $E_{i,j}$ variable.) Finally, cobra enforces the acyclicity of the searched-for compatible graph (that is, the graph whose edges are given by the $E_{i,j}$ that are set to True); Cobra does so by invoking a primitive provided by the solver.

Cobra vs. MonoSAT. One might ask: if cobra's encoding makes MonoSAT faster, why use MonoSAT? Can we take the domain knowledge further? Indeed, in the limiting case, cobra could re-implement the solver! However, MonoSAT, as an SMT solver, leverages many prior optimizations. One way to understand cobra's decomposition of function is that cobra's preprocessing exploits some of the structure created by the problem of verifying serializability, whereas the solver is exploiting residual structure common to many graph problems.

3.5 On strict serializability

Cobra's verifier checks strict serializability [56, 110] by adding *real-order edges* [38]—which capture the order of non-overlapping transactions in real time—to the known graph. The verifier then performs the serializability checking algorithm of Figure 3; as a result, the serialization order (in the searched-for compatible graph) respects the real-time order.

To get real-order edges, the verifier needs *timestamps* for each operation. The verifier can get them either from the database if it exposes the relevant interface (for example, Google Spanner [69]) or else from cobra's collectors. A naive way to go from timestamps to real-order edges is to examine every pair of transactions, and create a real-order edge when one transaction's commit timestamp is less than another's start timestamp. But this approach runs in time quadratic in the number of transactions. Instead, cobra borrows a prior algorithm [131, Fig. 6], which materializes the time precedence partial order in time $O(n+z)$, where n is the number of transactions and z is the minimum number of real-order edges needed.

A challenge is that clock drift in collectors makes it unsafe to infer real-time precedence relationships from timestamps. To tackle this problem, cobra introduces a *clock drift threshold* (100ms [15] by default). Cobra assumes that clock differences among collectors do not exceed this threshold; if they do, cobra may falsely reject a serializable history. With this assumption, cobra increases transactions' commit timestamps by the threshold. Thus, if two transactions have a real-order edge, one's original commit timestamp is earlier than the other transaction's start timestamp by at least the clock drift threshold. As a consequence, all transactions within a clock drift threshold are concurrent. Within such an interval, the verifier faces the computational expense that exists when there are no real-order edges, which calls for cobra's techniques (§3.1–§3.3) to accelerate verification (see §6.1 for relevant experiments).

4 Garbage collection and scaling

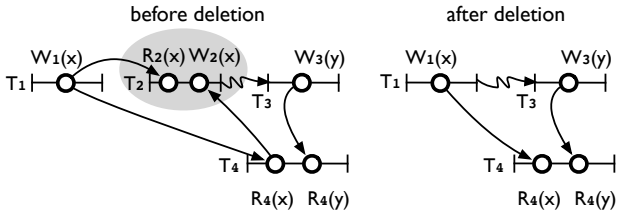
Cobra verifies in rounds. There are two motivations for rounds. First, new history is continually produced, of course. Second, there are limits on the maximum problem size (number of transactions) that the verifier can handle (§6.2); breaking the task into rounds keeps each solving task manageable.

In the first round, a verifier starts with nothing and creates a graph from CreateKnownGraph, then does verification. After that, the verifier receives more client histories; it reuses the graph from the last round (the g in ConstructEncoding, Figure 3, line 5), and adds new nodes and edges to it from the new history fragments received (Figure 2).

The technical problem is to keep the input to verification bounded. So the question cobra must answer is: which transactions can be deleted safely from history? Below, we describe the challenge (§4.1), the core mechanism of fence transactions (§4.2), and how the verifier deletes safely (§4.3). In this section, we describe the general rules and insights; a complete specification and correctness proof are in Appendix C [132].

4.1 The challenge

The core challenge is that past transactions can be relevant to future verifications; specifically, deleting a past transaction could cause the verifier to overlook future cycles.



Suppose a verifier saw three transactions (T_1, T_2, T_3) and wanted to remove T_2 (the shaded transaction) from consideration in future rounds. Later, the verifier observes a new transaction T_4 that violates serializability (and a fortiori, strict serializability) by reading from T_1 and T_3. To see the violation, notice that T_2 is logically subsequent to T_4, which generates a cycle ($T_4 \rightarrow T_2 \rightsquigarrow T_3 \rightarrow T_4$). Yet, if we remove T_2, there is no cycle. Hence, removing T_2 is not safe: future verifications would fail to detect certain kinds of serializability violations.

Note that this example does not require malicious or exotic behavior. For example, consider a geo-replicated database: a client can retrieve a stale version from a local replica.

4.2 Epochs and fence transactions

Cobra addresses this challenge by creating *epochs* that impose a coarse-grained ordering on transactions; the verifier can then discard information from older epochs. To avoid confusion, note that epochs are a separate notion from rounds: a verification round includes multiple epochs.

To memorialize epoch boundaries in history, clients issue *fence transactions*. A fence transaction is a transaction that reads-and-writes a single key named "EPOCH" (a dedicated

key that is used by fence transactions only). Each client issues fence transactions periodically (for example, every 20 transactions).

What prevents the database from defeating the point of epochs by placing all of the fence transactions at the beginning of a notional serial schedule? COBRA leverages a property of practical serializable databases: preserved *session order*. That is, the serialization order must obey the execution order within each session (defined in §2.1). Many production databases (for example, PostgreSQL, Azure Cosmos DB, and Google Cloud Datastore) provide this property; for those which do not, COBRA requires clients to build the session order, for example, by mandating that all transactions from the same session include a read-modify-write to a distinguished (per-session) key. Since transactions' serialization order obeys the session order, the epoch ordering intertwines with the workload. Indeed, the verifier adds to the known graph *session-order edges* (Figure 3, line 24), which capture the transaction issuing order in each session; the verifier gets that per-session order from collectors, which observe it directly.

The verifier also assigns an *epoch number* to each transaction. To do so, the verifier traverses the known graph (g), locates all the fence transactions, chains them into a list based on the RMW relation (§3.1), and assigns their position in the list as their epoch numbers. Then, the verifier scans the graph again, and for each normal transaction in a session that is between fences with epoch i and epoch j ($j \geq i+1$), the verifier assigns epoch number $j-1$.

During the scan, the verifier keeps track of the largest epoch number that has been seen or surpassed by every session, called $epoch_{agree}$. Then we have the following guarantee.

Guarantee. For any transaction T_i whose epoch $\leq (epoch_{agree}-2)$, and for any transaction (including future ones) T_j whose epoch $\geq epoch_{agree}$, the known graph g contains a path $T_i \rightsquigarrow T_j$.

To see why the guarantee holds, consider the path in three parts. First, for the fence transaction with epoch number $epoch_{agree}$ (denoted F_{ea}), g must have a path $F_{ea} \rightsquigarrow T_j$, through session-order edges. Similarly, for the fence transaction after T_i issued by the same session (denoted $F_{ea-\Delta}$), g has $T_i \rightsquigarrow F_{ea-\Delta}$. Finally, T_i has epoch $\leq (epoch_{agree}-2)$, so $F_{ea-\Delta}$ must have epoch $\leq (epoch_{agree}-1)$. Thus, $F_{ea-\Delta} \rightsquigarrow F_{ea}$ in g.

4.3 Garbage collection

COBRA takes a conservative approach. A transaction T can be safely deleted, if

(i) T has been *superseded*, meaning that no future transactions can precede T or directly succeed T in the known graph; and

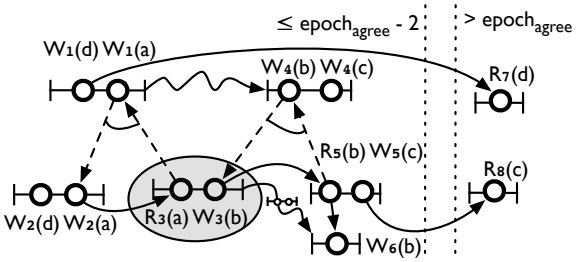
(ii) T is not involved in any potential cycle that includes edges from constraints whose resolution could be affected by future transactions.

Below, we delve into condition (i), then motivate condition (ii), and finally describe COBRA's procedure for garbage collection.

Identifying superseded transactions. Define the *frontier* as the set of transactions that contain the most recent writes to keys among transactions with epoch number $\leq (epoch_{agree}-2)$. The frontier captures the earliest transactions that future transactions can possibly read. A transaction T is *superseded* if: (1) T does not belong to the frontier, (2) T has epoch number $\leq (epoch_{agree}-2)$, and (3) for any transaction T' that has a path to T in the known graph, T' has epoch number $\leq (epoch_{agree}-2)$. Note that condition (2) does not subsume condition (3): we could have $T' \rightsquigarrow T$ with T' having a larger epoch than T (the Guarantee in §4.2 does not apply to transactions whose epochs differ by one).

At a high level, if a transaction T is superseded, the verifier can conclude that no future transactions should read from T; such a future transaction would have to be ordered before some frontier transaction, which makes a cycle by having a path back to the future transaction, per the Guarantee (§4.2). Thus T is a *candidate* to delete. However, being superseded is not a sufficient condition for safe deletion, as we illustrate next.

Superseded does not imply disposable. Here is an example:



The shaded transaction (T_3) is superseded (T_3 and its predecessor T_2 have epochs $\leq epoch_{agree}-2$, and T_3 does not belong to the frontier). Now consider the effect of future transactions T_7 and T_8. T_8 operates on key c; the other operations on this key are $W_4(c)$ and $W_5(c)$. By the guarantee (§4.2), both T_4 and T_5 happen before T_8. Plus, $R_8(c)$ reads from $W_5(c)$, hence $W_4(c)$ must happen before $W_5(c)$ (otherwise, $R_8(c)$ should have read from $W_4(c)$). Consequently, the constraint $\langle (T_5,T_4),(T_4,T_3)\rangle$, which arises from key b, is solved: $T_4 \rightarrow T_3$ is chosen. Similarly, because of $R_7(d)$, the other constraint (concerning key a) is solved and $T_3 \rightarrow T_1$. Thus, there is a cycle ($T_1 \rightsquigarrow T_4 \rightarrow T_3 \rightarrow T_1$). Yet, deleting T_3 would make the cycle undetectable.

The core issue here is that future transactions can affect the resolution of constraints among "old" transactions.

Identifying safe transactions. To garbage collect, the verifier clones the known graph (g in Fig. 3) into g'. Then, for each constraint (*con* in Fig. 3), the verifier adds all edges in both edge sets to g'. Finally, for each superseded transaction T, if T does not belong to any cycles in g' or belongs to cycles consisting only of superseded transactions, the verifier deletes T. This approach meets conditions (i) and (ii), as argued in Appendix C [132].

COBRA component	LOC written/changed
COBRA client library	
history recording	620 lines of Java
database adapters	900 lines of Java
COBRA verifier	
data structures and algorithms	2k lines of Java
GPU optimizations	550 lines of CUDA/C++
history parser and others	1.2k lines of Java

Figure 4: Components of COBRA implementation.

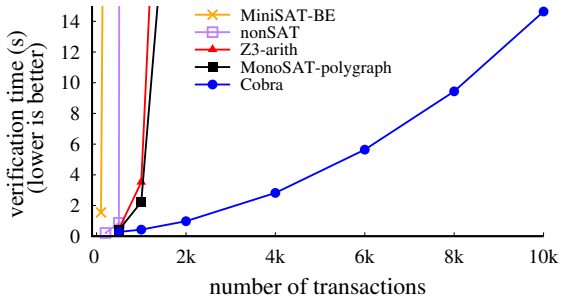


Figure 5: COBRA's running time is shorter than other baselines' on the BlindW-RW workload. The same holds on the other benchmarks (not depicted). Verification runtime grows superlinearly.

5 Implementation

Figure 4 lists the components of COBRA's implementation. COBRA's client library wraps other database libraries: JDBC, Google Datastore library, and RocksJava. It enforces the assumption of uniquely written values (§2.2), by adding a unique id to a client's writes, and stripping them out of reads. It also issues fence transactions (§4.2). Finally, we implement history collection (§2.1) in this library (the library writes operations to disk before sending them to the database); a better implementation would place this function in a proxy.

The verifier iterates the pruning logic within a round, stopping when it finds nothing more to prune or when it reaches a configurable maximum number of iterations (to bound the verifier's work); a better implementation would stop when the cost of the marginal pruning iteration exceeds the improvement in the solver's running time brought by this iteration.

Another aspect of pruning is GPU acceleration. Recall that pruning works by computing the transitive closure of the known edges (Figure 3, line 77). COBRA uses the standard algorithm: repeated squaring of the Boolean adjacency matrix [70, Ch.25] as long as the matrix keeps changing, up to $\log|V|$ matrix multiplications. ($\log|V|$ is the worst case and occurs when two nodes are connected by a $(\geq |V|/2+1)$-step path; at least in our experiments, this case does not arise much.) The execution platform is cuBLAS [12] (a dense linear algebra library on GPUs) and cuSPARSE [13] (a sparse linear algebra library on GPUs), which contain matrix multiplication routines.

COBRA includes several optimizations. It invokes a specialized routine for triangular matrix multiplication (after testing the graph for acyclicity and then indexing the vertices according to a topological sort, creating a triangular matrix). COBRA also exploits sparse matrix multiplication (cuSPARSE), and moves to ordinary (dense) matrix multiplication when the density exceeds a threshold (namely, "5% of the matrix elements are non-zero", the empirical cross-over point that we observed).

When COBRA's verifier detects a serializability violation, it creates a certificate with problematic transactions: either a cycle in the known graph (detected by COBRA's algorithms) or else a set of unsatisfiable clauses (produced by MonoSAT).

6 Experimental evaluation

We answer three questions:

- What are the verifier's costs and limits, and how do these compare to baselines?
- What is the verifier's end-to-end, round-to-round *sustainable capacity*? This determines the offered load (on the actual database) that the verifier can support.
- How much runtime overhead (in terms of throughput and latency) does COBRA impose for clients? And what are COBRA's storage and network overheads?

Benchmarks and workloads. We use four benchmarks:

- *TPC-C* [31] is a standard. A warehouse has 10 districts with 30k customers. There are five types of transactions (frequencies in parentheses): new order (45%), payment (43%), order status (4%), delivery (4%), and stock level (4%). In our experiments, we use one warehouse, and clients issue transactions based on the frequencies.
- *C-Twitter* [8] is a simple clone of Twitter, according to Twitter's own description [8]. Users can tweet a new post, follow/unfollow other users, and show a timeline (the latest tweets from followed users). Our experiments include 1000 users. Each user tweets 140-word posts and follows/unfollows other users based on a Zipfian distribution ($\alpha = 100$).
- *C-RUBiS* [30, 41] simulates bidding systems like eBay [30]. Users can register accounts, register items, bid for items, and comment on items. We initialize the market with 20k users and 200k items.
- *BlindW* measures COBRA's performance in extreme scenarios, specifically those with many blind writes (that is, writes not preceded by a read of the same key in the same transaction); this pattern is the fundamental source of uncertainty in constraints (§3). This benchmark creates 10k keys, and runs read-only and write-only transactions, each with eight operations. It has three variants: (1) *BlindW-RM* (Read Mostly), with 90% read-only transactions; (2) *BlindW-RW* (Read-Write), evenly divided between read-only and write-only transactions; and (3) *BlindW-WM* (Write Mostly), with 90% write-only transactions.

Violation	Database	#Txns	Time
G2-anomaly [19]	YugaByteDB 1.3.1.0	37.2k	66.3s
Disappearing writes [1]	YugaByteDB 1.1.10.0	2.8k	5.0s
G2-anomaly [18]	CockroachDB-beta 20160829	446	1.0s
Read uncommitted [26]	CockroachDB 2.1	20*	1.0s
Read skew [25]	FaunaDB 2.5.4	8.2k	11.4s

Figure 6: Serializability violations that COBRA checks. "Violation" describes the phenomenon that clients experience. "Database" is the database (with version number) that causes the violation. "#Txns" is the size of the violation history. "Time" is the runtime for COBRA to detect the violation.
* The bug report only contains a small fragment of the history.

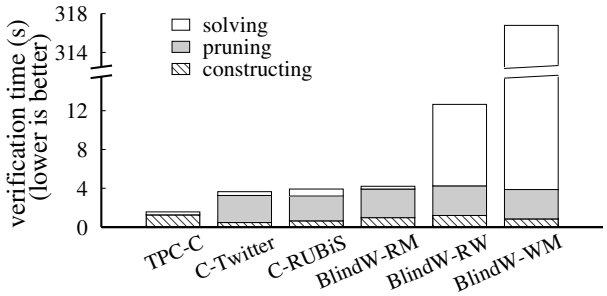


Figure 7: Decomposition of COBRA runtime, on 10k-transaction workloads. Pruning dominates for read-mostly workloads, whereas solving dominates for workloads with many writes.

Databases and setup. We evaluate COBRA on Google Cloud Datastore [21], RocksDB [29, 74] (both provide a key-value API), and PostgreSQL [28, 114] (which only supports the SQL interface, COBRA translates SQL queries to key-value operations). In our experimental setup, clients interact with Google Cloud Datastore through the wide-area Internet, and connect to a PostgreSQL server through a local 1 Gbps network. One client starts one session.

Database clients run on two machines with a 3.3GHz Intel i5-6600 (4-core) CPU, 16GB memory, a 250GB SSD, and Ubuntu 16.04. For PostgreSQL, a database instance runs on a machine with a 3.8GHz Intel Xeon E5-1630 (8-core) CPU, 32GB memory, a 1TB disk, and Ubuntu 16.04. For RocksDB, the same machine hosts the client threads and RocksDB threads, which all run in the same process. We use a *p3.2xlarge* Amazon EC2 instance as the verifier, with an NVIDIA Tesla V100 GPU, a 8-core CPU, and 64GB memory.

6.1 One-shot verification

In this section, we consider "one-shot verification": a verifier gets a history and decides whether that history is serializable. In our setup, clients record history fragments and store them as files; a verifier reads them from the local file system. In this section, the database is RocksDB (PostgreSQL and Google Cloud Datastore give similar results).

Baselines. We have four baselines:

- **A non-SAT serializability-checking algorithm ("nonSAT"):** To the best of our knowledge, the most efficient work for checking serializability that is not based on SAT or SMT solving is Biswas and Enea [59]. In our experiments, we use their Rust implementation [58].
- **SAT solver ("MiniSAT-BE"):** We use the same solving baseline that Biswas and Enea use for their own comparisons [59]: encoding serializability verification into SAT formulas, and feeding this encoding to MiniSAT [76], a popular SAT solver.
- **COBRA, subtracted ("MonoSAT-polygraph"):** We implement the original polygraph (§2.3), directly encode the constraints (without the techniques of §3), and feed them to the MonoSAT SMT solver [52].
- **SMT solver ("Z3-arith"):** An alternative use of SMT, and a natural baseline, is a linear arithmetic encoding: each node is assigned a distinct integer index, with read-from relationships creating inequality constraints, and writes inducing additional constraints (for a total of $O(|V|^2)$ constraints, as in §2.3). The solver is then asked to map nodes to integers, subject to those constraints [80, 91]. We use Z3 [73] as the solver (experiments below use Z3's default configuration; we also experimented with all four builtin linear integer arithmetic tactics, which produce similar results).

As a special case, there is an alternative baseline for TPC-C that has the same performance as COBRA and beats other baselines. Namely, for RMW transactions, add inferred read-dependency and write-dependency edges to a candidate graph (without constraints, so potentially missing dependency information), topologically sort it, and check whether the result matches history; if not, repeat. This process has even worse order complexity than the brute-force approach (§2.3). However, it works for TPC-C because that workload has *only* RMW transactions. Effectively, all of history coalesces to a single, correctly-ordered chain (§3.1), yielding a serialization graph.

In the experiments below, the baselines and COBRA make use of session order edges (§4.2; also called *program order* in BE [59] and its implementation [58]).

Verification runtime vs. number of transactions. We compare COBRA to other baselines, on the various workloads. We use 24 clients. We vary the number of transactions in the workload, and measure the verification time. Figure 5 depicts the results on the BlindW-RW benchmark. On all five benchmarks, COBRA does better than MonoSAT-polygraph and Z3-arith, which do better than MiniSAT-BE and nonSAT.

Detecting serializability violations. We investigate COBRA's performance on unsatisfiable instances: does COBRA search for an unacceptably long time on real-world workloads? We consider five workloads that are known to have serializability violations [1, 18, 19, 25, 26]. We experiment by downloading the reported histories from their bug repositories and feeding them to COBRA's verifier. Figure 6 shows the results. COBRA detects all violations and finishes in reasonable time.

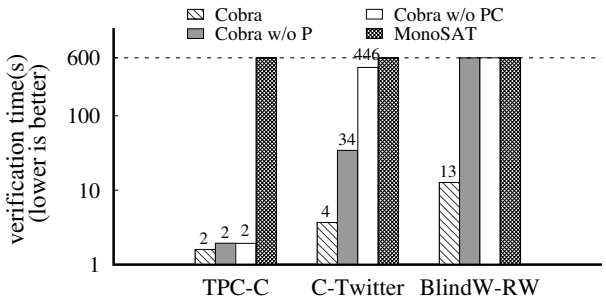


Figure 8: Differential analysis on several workloads, log-scale, with runtime above bars. Experiments time out at 10min (dotted line); no runtime is shown for timed-out experiments. On TPC-C, combining writes exploits the RMW pattern and solves all the constraints. On C-Twitter, each of COBRA's components contributes meaningfully. On BlindW-RW, pruning is essential, because the workload has many blind writes which cannot benefit from the other two techniques.

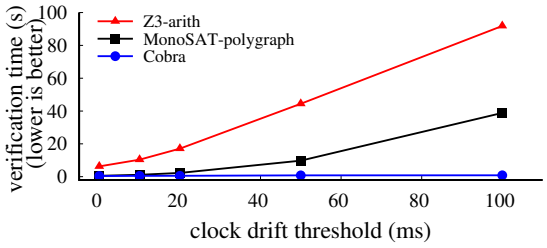


Figure 9: COBRA's running time is shorter than other baselines' on checking strict serializability under clock drift. The workload is 2,000 transactions of BlindW-RW (clock drift threshold of 100 ms).

Decomposition of COBRA's verification runtime. We measure the wall clock time of COBRA's verification, broken into stages: *constructing*, which includes creating the known graph, combining writes, and creating constraints (§3.1–§3.2); *pruning* (§3.3), which includes the time taken by the GPU; and *solving* (§3.4), which includes the time spent within MonoSAT. We experiment with all benchmarks, with 10k transactions.

Figure 7 depicts the results. In benchmarks with RMWs only (the left one), there are no constraints, so COBRA doesn't prune (see also the special case baseline, §6.1). In benchmarks with many reads and RMWs (the second to fourth bars), the dominant component is pruning not solving, because COBRA's own logic identifies concrete dependencies. In benchmarks with many blind writes (the last two), solving is a much larger contributor because COBRA cannot eliminate as many constraints, leading to a larger search space, an effect that grows more pronounced as the fraction of blind writes increases. On the other hand, a majority of writes is not consistent with the patterns in common online transaction processing workloads (OLTP), where reads dominate.

Differential analysis. We experiment with four variants: COBRA itself; COBRA without pruning (§3.3); COBRA without pruning and coalescing (§3.2), which is equivalent to MonoSAT plus write combining (§3.1); and the MonoSAT baseline.

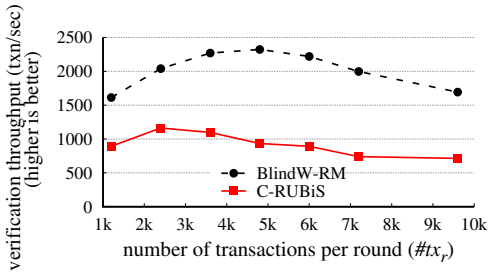


Figure 10: Verification throughput vs. round size ($\#tx_r$). The verification capacity for BlindW-RM (the dashed line) is 2.3k txn/sec when $\#tx_r$ is 5k; the capacity for C-RUBiS (the solid line) is 1.2k txn/sec when $\#tx_r$ is 2.5k.

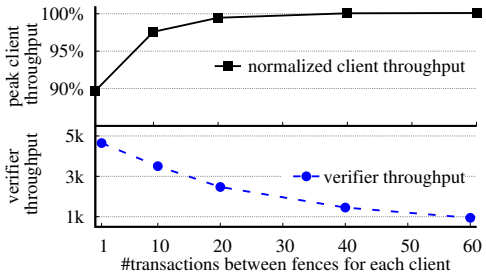


Figure 11: Client and verifier throughputs with different fence frequencies. Client throughput (the solid line) is normalized to the workload without fence transactions. In BlindW-RM, each normal transaction has 8 operations (§6), and fence transactions have 1–2 operations.

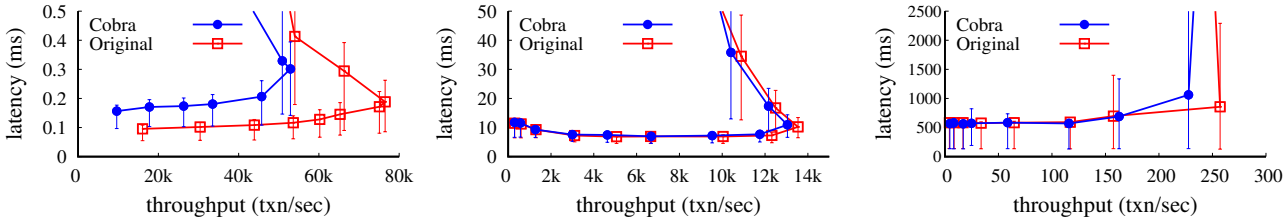
We experiment with three benchmarks, with 10k transactions. Figure 8 depicts the results.

Checking strict serializability under clock drift. Clock drift adds complexity to strict serializability (§1,§3.5). To measure this effect, we experiment with COBRA, MonoSAT-polygraph, and Z3-arith, under different clock drifts, on the same workload. The workload has eight clients running BlindW-RW on 1k keys for one second with a throughput of 2k transaction/sec. To control computational overhead, the clients issue 20 transactions every 10ms. The maximum clock drift threshold is 100 ms [15]; similar thresholds can be found elsewhere [10, 37]. Figure 9 depicts the results; COBRA outperforms the baselines by 45× and 107× in verification time.

6.2 Scaling

What offered load (to the database) can COBRA support on an ongoing basis? To answer this question, we must quantify COBRA's *verification capacity*, in txns/second. This depends on the characteristics of the workload, the number of transactions one round (§4) verifies ($\#tx_r$), and the average time for one round of verification (t_r). Note that the variable here is $\#tx_r$; t_r is a function of that choice. So the *verification capacity* for a particular workload is defined as: $\max_{\#tx_r}(\#tx_r/t_r)$.

To investigate this quantity, we run all our benchmarks on RocksDB with 24 concurrent clients, each configured to issue



(a) RocksDB (b) PostgreSQL (c) Google Datastore

Figure 12: Throughput and latency, for C-Twitter benchmark. For our RocksDB setup, 90th percentile latency increases by 2×, with 50% throughput penalty, an artifact of history collection (disk bandwidth contention between clients and the DB). COBRA imposes minor overhead for our PostgreSQL. For Google Datastore, the throughput penalty reflects a ceiling (a maximum number of operations per second) imposed by the cloud service and the extra operations caused by fence transactions.

fence transactions every 20 transactions. We generate a 100k-transaction history ahead of time. For that same history, we vary $\#tx_r$, plot $\#tx_r/t_r$, and choose the optimum.

Figure 10 depicts the results for two benchmarks (C-RUBiS and BlindW-RM); C-Twitter and TPC-C have similar results (not depicted), but BlindW-RW and BlindW-WM run out of memory (we elaborate below). When $\#tx_r$ is smaller, COBRA does not have enough transactions to garbage collect, hence wastes cycles on redundantly analyzing transactions from prior rounds; when $\#tx_r$ is larger, COBRA suffers from a problem size that is too large (recall that verification time increases superlinearly; §6.1).

History eventually exceeds GPU memory on the BlindW-RW and BlindW-WM benchmarks because blind writes limit COBRA's ability to garbage collect transactions: blind writes cannot benefit from combining writes (§3.1), hence many constraints remain, causing transactions to be involved in uncertain constraints, and thus not collectible (§4.3). Addressing this issue is future work (§8).

Fence frequency. The choice of fence frequency trades off verification capacity and peak client-side throughput. To quantify, we do the same BlindW-RM experiments as in Figure 10, this time fixing round size (at 5k transactions) and varying fence frequency.

Figure 11 depicts the results. The verifier has better throughput if clients issue fence transactions more frequently. The reason is that more fence transactions result in smaller epoch sizes, hence transactions can be garbage collected earlier, and the problem size for the verifier in each round is smaller. Moreover, with more fence transactions, the problem in each round is easier to solve because fence transactions add ordering constraints, which further reduce the number of possibly-valid execution schedules. However, more frequent fence transactions sacrifices peak client-side throughput because more resources are occupied by fence transactions.

The right setting of fence frequency depends on client offered load, peak:average throughput ratio, database capacity, and tolerance for latency. If the frequency is set too high (toward the left side of the x-axis), clients will no longer be able to offer the original workload with acceptable latency. On the other hand, for too-low frequencies (toward the right side of the x-axis), the verifier will not be able to keep up with the database's average load. Of course, if client load is constant, the fence frequency should be chosen as the point where verifier throughput equals client offered load.

workload	network overhead		history size
	traffic	percentage	
BWrite-RW	227.4 KB	7.28%	245.5 KB
C-Twitter	292.9 KB	4.46%	200.7 KB
C-RUBiS	107.5 KB	4.53%	148.9 KB
TPC-C	78.2 KB	2.17%	1380.8 KB

Figure 13: Network and storage overheads per 1k transactions. The network overheads comes from fence transactions and the metadata (transaction ids and write ids) added by COBRA's client library.

6.3 Online overheads

The baseline in this section is the legacy system; that is, clients use the unmodified database library (for example, JDBC), with no recording of history.

Latency-versus-throughput. We evaluate COBRA's client-side throughput and latency in the three setups, tuning the number of clients (up to 256) to saturate the databases. Figure 12 depicts the results. (Although these results include the overhead of collecting histories in the client library (§5), that overhead is negligible, as the log size is small and disk latency is lower than network latency.)

Network cost and history size. We evaluate the network traffic on the client side by tracking the number of bytes sent over the NIC. We measure the history size by summing sizes of the history files. Figure 13 summarizes.

6.4 Summary of experimental evaluation

COBRA improves by at least 10× on baselines in verification cost (Figure 5), detects real-world issues (Figure 6), gains from its techniques versus the baseline (Figures 7 and 8), and imposes tolerable overhead (Figures 12 and 13).

Furthermore, its sustained throughput of 2k txn/sec (Figure 10) corresponds to large-scale real-world workloads. While 2k/sec might not sound large, recall that the verifier's perfor-

mance requirement is to match *average* database load (§2.1). An average of 2k/sec corresponds to 170M/day, sufficient to handle Apple Pay [6] (33M txn/day), Visa [33] (150M txn/-day), and others. Of course, a "transaction", in the sense of a payment, might translate to several database transactions, so the comparison is inexact.

7 Related work

As stated earlier (§1), COBRA is the first system that verifies the executions of (a) black box databases, for (b) serializability, under (c) workloads of realistic scale.

Three works that we are aware of tackle (a) and (b) together. Biswas and Enea [59] was covered in Section 1 and compared in Section 6. Sinha et al. [124] use SMT solvers to analyze all possible interleavings for a concurrent program, to search for serializability violations. Finally, Gretchen [23] is an experimental checker that verifies the non-strict serializability of a COBRA-style history. Gretchen encodes the history as constraints [67] (similar to our MiniSAT-BE baseline; §6.1) and solves them with the fzn-gecode [20] solver.

A recent work that deserves special mention is Elle [94], which tests for isolation anomalies, and has found many isolation bugs in production databases. (Elle is part of the impactful Jepsen [14] project, which we discuss later in this section.) Elle has two modes for testing serializability. In one, it verifies Adya's serializability (PL-3 [38]), the same goal as COBRA. But Elle in this mode requires a workload that makes the version order (§2.2) manifest; for example, clients invoke "append", and writes become appends to a list (Elle in this mode also supports counters and sets). In relying on a specific API and a specific workload for testing, this mode does not meet our notion of black box (§1).

In the second mode, Elle works over arbitrary observations of key-value input/output, the same setup as COBRA. Without a determined version order, it applies heuristics to identify bugs. These heuristics are useful but not comprehensive, so this is not verification. For example, if a history contains a set of concurrent transactions that form a cycle through anti-dependencies, Elle's current heuristics do not detect the non-serializability.

Checking consistency. Serializability is a particular *isolation* level in a transactional system—the I in ACID transactions. In shared memory systems and systems that offer replication (but do not necessarily support transactions), there is an analogous correctness contract, namely *consistency*. (Confusingly, the "C(onsistency)" in ACID transactions refers to something else [47].) Example consistency models are linearizability [88], sequential consistency [97], and eventual consistency [112]. Testing adherence to these models is an analogous problem to ours. In both cases, one searches for a total order of operations that fits the ordering constraints of both the model and the history [82]. As in checking serializability, the computational complexity of checking consistency decreases if a stronger model is targeted (for example, linearizability vs. sequential consistency) [81], or if more ordering information can be (intrusively) acquired (by opening black boxes) [123, 139].

Concerto [43] uses *deferred verification*, allowing it to exploit offline memory checking [60] to check online the sequential consistency of a highly concurrent key-value store. Concerto's design achieves orders-of-magnitude performance improvement compared to Merkle tree-based approaches [60, 106], but it also requires modifying the storage layer. (See elsewhere [75, 98] for algorithms related to Concerto.)

A body of work examines cloud storage consistency [39, 42, 83, 101, 102, 115, 135, 142]. These works rely on extra ordering information obtained through techniques like loosely- or well-synchronized clocks [39, 42, 82, 83, 93, 102, 115, 135, 142], or client-to-client communication [101, 122], or by guessing [143] (which risks falsely rejecting honest executions). As another example, a gateway that sequences the requests can ensure consistency by enforcing ordering [90, 113, 122, 125], thereby dramatically reducing concurrency.

Some of COBRA's techniques are reminiscent of these works, such as its use of serialization graphs [42, 82]. However, a substantial difference is that COBRA neither modifies the "memory" (the database) to get information about the actual internal schedule nor depends on external synchronization. COBRA of course exploits epochs (§4.2), but this is for scaling, not core to the verification task, and invokes standard database interfaces.

Execution integrity. Our problem relates to the broad category of *execution integrity*—ensuring that a module in another administrative domain is executing as expected.

One approach is to use trusted components. For example, Byzantine fault tolerant (BFT) replication [66] (where the assumption is that a super-majority is not faulty) and TEEs (trusted execution environments, comprising TPM-based systems [68, 87, 104, 105, 111, 117, 119, 126] and SGX-based systems [44, 45, 51, 89, 95, 118, 121, 125]) ensure that the right code is running. However, this does not ensure that the code itself is right; concretely, if a database violates serializability owing to a bug, neither BFT nor SGX hardware helps.

Other examples are Verena [92], Orochi [131], AVM [85], and Ripley [134]. These systems provide end-to-end assurance that a whole stack is executing as it should, but they are not black box. COBRA is the other way around: it treats the database as a black box, but its purview is limited to the database.

A class of systems uses complexity-theoretic and cryptographic mechanisms [61, 120, 145, 146]. None of these works handle systems of realistic scale, and only one of them [120] handles concurrent workloads. An exception is Obladi [71], which remarkably provides ACID transactions atop an ORAM abstraction by exploiting a trusted proxy that carefully manages the interplay between concurrency control and the ORAM protocol; its performance is surprisingly good (as cryptographic-based systems go) but still pays 1-2 orders of magnitude over-

head in throughput and latency.

Detecting application anomalies caused by weak consistency. Several works [63, 109, 116] detect anomalies for applications deployed on weakly consistent storage. Like COBRA, these works use SAT/SMT solvers on graph-related problems. But the similarities end there: these works analyze application behavior, taking the storage layer as *trusted* input. As a consequence, the technical mechanisms are very different.

Testing distributed systems. There is a line of research on testing the correctness of distributed systems under various failures, including network partition [40], power failures [147], and storage faults [78]. Among these, Jepsen [14] is a very successful testing framework (the aforementioned Elle is one of Jepsen's checkers) with active, ongoing innovation, which has detected large numbers of correctness bugs in production distributed systems. COBRA is complementary (and intended to be complimentary) to these works. Indeed, COBRA uses several of Jepsen's traces in Figure 6 (§6.1).

Definitions and interpretations of isolation levels. COBRA of course uses dependency graphs, which are a common tool for reasoning about isolation levels [38, 56, 110]. However, isolation levels can be interpreted via other means such as excluding anomalies [53] and client-centric observations [72]; an open and intriguing question is whether the other definitions yield a more intuitive or more easily-implemented encoding and algorithm than the one in COBRA.

8 Discussion, future work, and conclusion

Applicability. COBRA cannot prevent misbehavior, only detect it. On the other hand, *no* system that we are aware of can detect and prevent serializability violations online. Meanwhile, COBRA could contribute to recovery: given a certificate (§5), the user could supply a candidate serialization order, enabling roll back and replay. See also Concerto's eloquent case for deferred verification [43, §1.1].

Who would use COBRA? We covered some scenarios in Section 2.1. Another is to use COBRA as the checker of a testing framework (for example, Jepsen [14], §7). Then one could insert malfunctions into various layers of the system (OS, storage, network) and avoid instrumenting the database.

One might assume that the verifier needs to be at least as powerful as the database, so why have the database? While they must match in long-term average transactions/sec (§2.1), the two do different kinds of work per transaction. The database provides geo-replication, concurrency control, crash-atomicity, durability, load-balancing, and more; the verifier is a single machine and purely algorithmic.

Limitations and future work. COBRA can be slow for certain workloads (for example, when there are many unconstrained writes, as in the BlindW-WM benchmark; §6.1). In fact, COBRA's worst-case running time is in principle exponential; future work is to investigate whether there are real-world workloads that induce this behavior, or does it just happen under contrived problem instances as in the NP-reduction?

Consistent with our experiments, we expect "real-world" workloads not to trigger this behavior. For intuition, low contention on each key yields a relatively small number of constraints and a small search space; the extreme is that each key is touched once, yielding no dependencies. If there is high contention with sufficient reads, there are more dependencies among transactions, which imposes more ordering. An extreme case is that there is only one key, and transactions read and write this key, so that all transactions are ordered accordingly.

We have assumed that the verifier and the collectors operate fault-free. Future work is to make them fault-tolerant. To that end, COBRA could use standard techniques (for example, transparent state machine replication) or extend its protocols to handle failures. Note that even if some history fragments are lost, COBRA can (with minor modifications) produce meaningful results: a cyclic dependency (serializability violation) in a partial history is also a violation against the full history. Another idea is to use COBRA to infer what the missing transactions would have to be in order to ensure serializability.

COBRA focuses on serializability and strict serializability; future work is extending to other isolation levels. Relatedly, COBRA does not support range queries and other high-level operators (for example, sum and join); if applications want them, they have to rewrite queries (§1). Handling these queries "natively" would require the verifier to analyze both keys that are returned and keys that are *not* returned.

Making garbage collection more aggressive is another area of potential improvement, for example, by allowing the verifier to query the database to resolve certain constraints.

Conclusion. A final critique is that we lack a sensational headline, as we did not identify novel serializability violations. However, validation doesn't always produce a gotcha: from our perspective, it's equally significant to be able to report on a system that gives us confidence that cloud databases do meet serializability. This was something we used to have to *trust*; COBRA, however imperfect, helps us be *sure*.

Acknowledgments

Sebastian Angel, Miguel Castro, Pete Chen, Byron Cook, Andreas Haeberlen, Dennis Shasha, Ioanna Tzialla, Thomas Wies, and Lingfan Yu made helpful comments and gave useful pointers. We thank the anonymous reviewers (including at SOSP and NSDI) for careful and constructive comments, and likewise our shepherd Chris Hawblitzel. We thank the anonymous artifact evaluators for their patience and attention to detail. This work was supported by NSF grants CNS-1423249 and CNS-1514422, ONR grant N00014-16-1-2154, AFOSR grants FA9550-15-1-0302 and FA9550-18-1-0421, and DARPA under Agreement HR00112020022.

References

[1] Acknowledged inserts can be present in reads for tens of seconds, then disappear. https://github.com/YugaByte/yugabyte-db/issues/824.

[2] Amazon Aurora. https://aws.amazon.com/rds/aurora/.

[3] Amazon Aurora MySQL Reference. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraMySQL.Reference.html.

[4] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[5] Amazon DynamoDB Transactions. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/transaction-apis.html.

[6] Apple pay transaction volume and new user growth outpacing paypal, tim cook says. https://9to5mac.com/2019/07/30/apple-pay-transactions-users-paypal/.

[7] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/.

[8] Big data in real time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter.

[9] CockroachDB: Distributed SQL. https://www.cockroachlabs.com.

[10] CockroachDB: What happens when node clocks are not properly synchronized? https://www.cockroachlabs.com/docs/stable/operational-faqs.html#what-happens-when-node-clocks-are-not-properly-synchronized.

[11] CockroachDB's consistency model. https://www.cockroachlabs.com/blog/consistency-model/.

[12] cuBLAS: Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas.

[13] cuSPARSE: Sparse Linear Algebra on GPUs. https://developer.nvidia.com/cusparse.

[14] Distributed system safety research. https://jepsen.io/.

[15] Executive summary: Computer network time synchronization. https://www.eecis.udel.edu/~mills/exec.html.

[16] FaunaDB. https://fauna.com.

[17] FoundationDB. https://www.foundationdb.org.

[18] G2: anti-dependency cycles. https://github.com/cockroachdb/cockroach/issues/10030.

[19] G2-item anomaly with master kills. https://github.com/YugaByte/yugabyte-db/issues/2125.

[20] Gecode: Flatzinc. https://www.gecode.org/flatzinc.html.

[21] Google Cloud Datastore. https://cloud.google.com/datastore/.

[22] Google Cloud Spanner. https://cloud.google.com/spanner/.

[23] Gretchen: Offline serializability verification, in clojure. https://github.com/aphyr/gretchen.

[24] How Halo 5 implemented social gameplay using Azure Cosmos DB. https://azure.microsoft.com/en-us/blog/how-halo-5-guardians-implemented-social-gameplay-using-azure-documentdb/.

[25] Jepsen: Faunadb 2.5.4. http://jepsen.io/analyses/faunadb-2.5.4.

[26] Lessons learned from 2+ years of nightly jepsen tests. https://www.cockroachlabs.com/blog/jepsen-tests-lessons/.

[27] Norwegian electronics giant scales for sales, sets record with cloud-based transaction processing. https://customers.microsoft.com/en-us/story/elkjop-retailers-azure.

[28] PostgreSQL. https://www.postgresql.org/.

[29] RocksDB. https://rocksdb.org/.

[30] RUBiS. https://rubis.ow2.org/.

[31] TPC-C. http://www.tpc.org/tpcc/.

[32] Transactions, cloud Spanner. https://cloud.google.com/spanner/docs/transactions.

[33] Visa: Small business retail. https://usa.visa.com/run-your-business/small-business-tools/retail.html.

[34] The Yices SMT solver. http://yices.csl.sri.com/.

[35] YugaByte db 1.3.1, undercounting counter. http://jepsen.io/analyses/yugabyte-db-1.3.1.

[36] YugaByte DB: Home. https://www.yugabyte.com.

[37] yugabyte source code. https://github.com/yugabyte/yugabyte-db/blob/3b90e8560b8d8bc81fba6ba9b9f2833e83e2244e/src/yb/util/physical_time.cc#L36.

[38] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[39] A. S. Aiyer, E. Anderson, X. Li, M. A. Shah, and J. J. Wylie. Consistability: Describing usually consistent systems. In *Proc. HotDep*, Dec. 2008.

[40] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proc. OSDI*, Oct. 2018.

[41] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *Proc. IEEE WWC*, Nov. 2002.

[42] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. What consistency does your key-value store *actually* provide? In *Proc. HotDep*, Oct. 2010. Full version: Technical Report HPL-2010-98, Hewlett-Packard Laboratories, 2010.

[43] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: a high concurrency key-value store with integrity. In *Proc. SIGMOD*, May 2017.

[44] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, et al. SCONE: Secure Linux containers with Intel SGX. In *Proc. OSDI*, Oct. 2016.

[45] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch. LibSEAL: Revealing service integrity violations using trusted execution. In *Proc. EuroSys*, Apr. 2018.

[46] A. Awad and B. Karp. Execution integrity without implicit trust of system software. In *ACM Workshop on System Software for Trusted Execution (SysTEX)*, 2019.

[47] P. Bailis. Linearizability versus serializability. http://www.bailis.org/blog/linearizability-versus-serializability/, Sept. 2014.

[48] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: virtues and limitations. *PVLDB*, Sept. 2014.

[49] T. Balyo, M. J. Heule, and M. Jarvisalo. SAT competition 2016: Recent developments. In *Proc. AAAI*, Feb. 2017.

[50] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. CAV*, July 2011.

[51] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proc. OSDI*, Oct. 2014.

[52] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. In *Proc. AAAI*, Jan. 2015.

[53] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD*, May 1995.

[54] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[55] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[56] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *TSE*, SE-5(3), May 1979.

[57] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[58] R. Biswas and C. Enea. dbcop source code. https://zenodo.org/record/3367334.

[59] R. Biswas and C. Enea. On the complexity of checking transactional consistency. In *Proc. OOPSLA*, Oct. 2019.

[60] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2-3), Sept. 1994.

[61] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proc. SOSP*, Nov. 2013.

[62] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *Proc. POPL*, Jan. 2017.

[63] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Static serializability analysis for causal consistency. In *Proc. PLDI*, 2018.

[64] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *Proc. CAV*, July 2008.

[65] M. A. Casanova. *The concurrency control problem for database systems*. Number 116. Springer Science & Business Media, 1981.

[66] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. OSDI*, Feb. 1999.

[67] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015)*, Sept. 2015.

[68] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar. Towards verifiable resource accounting for outsourced computation. In *Proc. VEE*, Mar. 2013.

[69] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *TOCS*, 31(3), June 2013.

[70] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, third edition*. The MIT Press, 2009.

[71] N. Crooks, M. Burke, E. Cecchetti, S. Harel, L. Alvisi, and R. Agarwal. Obladi: Oblivious serializable transactions in the cloud. In *Proc. OSDI*, Oct. 2018.

[72] N. Crooks, Y. Pu, L. Alvisi, and A. Clement. Seeing is believing: a client-centric specification of database isolation. In *Proc. PODC*, July 2017.

[73] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, Mar. 2008.

[74] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in RocksDB. In *Proc. CIDR*, Jan. 2017.

[75] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *Proc. TCC*, Mar. 2009.

[76] N. Eén and N. Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*. Springer, 2003.

[77] A. Fekete, S. N. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. *Proceedings of the VLDB Endowment*, 2(1):467–478, 2009.

[78] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Proc. FAST*, Feb. 2017.

[79] M. Gebser, T. Janhunen, and J. Rintanen. Answer set programming as SAT modulo acyclicity. In *Proc. ECAI*, 2014.

[80] M. Gebser, T. Janhunen, and J. Rintanen. SAT modulo graphs: acyclicity. In *Proc. JELIA*, 2014.

[81] P. B. Gibbons and E. Korach. Testing shared memories. *SIJC*, 26(4), Aug. 1997.

[82] W. Golab, X. Li, and M. Shah. Analyzing consistency properties for fun and profit. In *Proc. PODC*, June 2011.

[83] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *Proc. ICDCS*, June 2014.

[84] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proc. DATE*, Mar. 2002.

[85] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proc. OSDI*, Oct. 2010.

[86] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *Proc. ICSE*, May 2008.

[87] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: end-to-end security via automated full-system verification. In *Proc. OSDI*, Oct. 2014.

[88] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), July 1990.

[89] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: a distributed sandbox for untrusted computation on secret data. In *Proc. OSDI*, Oct. 2016.

[90] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *Proc. ICDE*, Apr. 2013.

[91] M. Janota, R. Grigore, and V. M. Manquinho. On the quest for an acyclic graph. *CoRR*, abs/1708.01745, Aug. 2017.

[92] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for Web applications. In *Proc. S&P*, May 2016.

[93] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Proc. S&P*, May 2015.

[94] K. Kingsbury and P. Alvaro. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554*, 2020.

[95] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proc. EuroSys*, Apr. 2018.

[96] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proc. EuroSys*, Apr. 2013.

[97] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *TC*, C-28(9), Sept. 1979.

[98] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. SIGMOD*, June 2006.

[99] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *Proc. AAAI*, Feb. 2016.

[100] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proc. SIGMOD*, May 2017.

[101] Q. Liu, G. Wang, and J. Wu. Consistency as a service: Auditing cloud consistency. *TNSM*, 11(1), Mar. 2014.

[102] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *Proc. SOSP*, Oct. 2015.

[103] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9), July 2013.

[104] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proc. S&P*, May 2010.

[105] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. EuroSys*, Apr. 2008.

[106] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proc. Crypto*, Aug. 1987.

[107] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*, June 2001.

[108] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proc. OSDI*, Oct. 2014.

[109] K. Nagar and S. Jagannathan. Automated detection of serializability violations under weak consistency. *arXiv preprint arXiv:1806.08416*, 2018.

[110] C. H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4), Oct. 1979.

[111] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping trust in modern computers*. Springer, 2011.

[112] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP*, Oct. 1997.

[113] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proc. USENIX ATC*, June 2011.

[114] D. R. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *PVLDB*, 5(12), Aug. 2012.

[115] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie. Toward a principled framework for benchmarking consistency. In *Proc. HotDep*, Oct. 2012.

[116] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan. Clotho: directed test generation for weakly consistent database systems. In *Proc. OOPSLA*, Oct. 2019.

[117] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proc. USENIX Security*, Aug. 2004.

[118] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. S&P*, May 2015.

[119] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proc. SOSP*, Oct. 2005.

[120] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proc. OSDI*, Oct. 2018.

[121] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In *Proc. NDSS*, Feb. 2017.

[122] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. CCSW*, Oct. 2010.

[123] A. Sinha and S. Malik. Runtime checking of serializability in software transactional memory. In *Proc. IPDPS*, Apr. 2010.

[124] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predicting serializability violations: SMT-based search vs. DPOR-based search. In *Haifa Verification Conference*, 2011.

[125] R. Sinha and M. Christodorescu. VeritasDB: High throughput key-value store with integrity. *IACR Cryptology ePrint Archive*, 2018.

[126] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proc. SOSP*, Oct. 2011.

[127] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. SAT*, June 2009.

[128] A. Stump, C. W. Barrett, and D. L. Dill. CVC: a cooperating validity checker. In *Proc. CAV*, July 2002.

[129] C. Su, N. Crooks, C. Ding, L. Alvisi, and C. Xie. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, May 2017.

[130] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *Proc. RV*, Sept. 2011.

[131] C. Tan, L. Yu, J. Leners, and M. Walfish. The efficient server audit problem, deduplicated re-execution, and the web. In *Proc. SOSP*, Oct. 2017.

[132] C. Tan, C. Zhao, S. Mu, and M. Walfish. Cobra: Making transactional key-value stores verifiably serializable (extended version). arXiv:1912.09018, https://arxiv.org/abs/1912.09018, Dec. 2019.

[133] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora : Design considerations for high throughput cloud-native relational databases. In *Proc. SIGMOD*, May 2017.

[134] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proc. CCS*, Nov. 2009.

[135] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *Proc. CIDR*, Jan. 2011.

[136] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proc. SIGMOD*, June 2016.

[137] T. Warszawski and P. Bailis. ACIDRain: Concurrency-related attacks on database-backed web applications. In *Proc. SIGMOD*, May 2017.

[138] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[139] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *JPDC*, 17(1-2), Jan. 1993.

[140] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *Proc. SOSP*, Oct. 2015.

[141] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Notices*, 40(6), 2005.

[142] K. Zellag and B. Kemme. How consistent is your cloud application? In *Proc. SoCC*, Oct. 2012.

[143] K. Zellag and B. Kemme. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *The VLDB Journal*, 23(1), Feb. 2014.

[144] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. SOSP*, Oct. 2015.

[145] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *Proc. S&P*, May 2017.

[146] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *Proc. CCS*, Oct. 2015.

[147] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proc. OSDI*, Oct. 2014.

A Artifact Appendix

This artifact contains two parts: a COBRA verifier and COBRA clients. The COBRA verifier checks serializability of a set of transactions (called a history). COBRA clients include database clients and COBRA's client library. Database clients are benchmark programs that interact with a black-box database (not part of COBRA) and generate histories. COBRA's client library wraps database libraries, encodes and decodes values to and from the database, and records histories to logs.

COBRA's artifact, including source code and comprehensive instructions for running the code and reproducing results, is released at: https://github.com/DBCobra/CobraHome. COBRA's verifier requires an NVIDIA GPU to run, and COBRA depends on Linux (tested on Ubuntu 18.04), Java (1.8 or higher), CUDA (tested on 10.0.130), and MonoSAT (1.6.0).



Determinizing Crash Behavior with a Verified Snapshot-Consistent Flash Translation Layer

Yun-Sheng Chang Yao Hsiao Tzu-Chi Lin Che-Wei Tsao Chun-Feng Wu
Yuan-Hao Chang Hsiang-Shang Ko Yu-Fang Chen

Institute of Information Science, Academia Sinica, Taiwan

Abstract

This paper introduces the design of a snapshot-consistent flash translation layer (SCFTL) for flash disks, which has a stronger guarantee about the possible behavior after a crash than conventional designs. More specifically, the flush operation of SCFTL also has the functionality of making a "disk snapshot." When a crash occurs, the flash disk is guaranteed to recover to the state right before the last flush. The major benefit of SCFTL is that it allows a more efficient design of upper layers in the storage stack. For example, the file system built on SCFTL does not require the use of a journal for crash recovery. Instead, it only needs to perform a flush operation of SCFTL at the end of each atomic transaction. We use a combination of a proof assistant, a symbolic executor, and an SMT solver, to formally verify the correctness of our SCFTL implementation. We modify the xv6 file system to support group commit and utilize SCFTL's stronger crash guarantee. Our evaluation using file system benchmarks shows that the modified xv6 on SCFTL is 3 to 30 times faster than xv6 with logging on conventional FTLs, and is in the worst case only two times slower than the state-of-the-art setting: the ext4 file system on the Physical Block Device (pblk) FTL.

1 Introduction

In modern computer systems, data storage usually needs to go through multiple layers, starting from a specific application, going through the file system, and eventually reaching the physical device. Usually, we refer to those layers as the *storage stack*. The design of a correct storage stack is non-trivial. For instance, in order to maintain efficiency, I/O operations sending from one layer can be reordered to reduce the overall waiting time. More importantly, under a sudden power loss or a system crash, a storage system should correctly recover the stored data. Due to the high complexity of the system design, in recent years, a significant amount of research effort is devoted to applying formal methods to provide rigorous correctness guarantees about storage stacks. Among those, one crucial direction is proving crash safety [3, 9, 10, 12, 13, 42].

Crash recovery is a critical issue; no one wants to lose important data after one accidental power loss. In the state-of-the-art system design, each component in the storage stack has its crash recovery mechanism and usually makes only minimal assumptions about its lower layers. For instance, the file system usually assumes the underlying physical device follows the *asynchronous disk model*. The model provides only minimal guarantees about its possible behavior when the system crashes. As a result, the file system needs to implement a heavyweight crash recovery mechanism.

The more recent design of physical devices offers much stronger guarantees while maintaining similar performance. For instance, the *prefix-preserving* disk model [11] guarantees to recover to some state after the last flush operation without operation reordering. The *snapshot-consistent* disk model we propose in this paper provides an even stronger guarantee. The advance in the physical device design provides us with an excellent opportunity to rethink the design of the entire storage stack. The stronger guarantees of the physical device enable a cleaner and more efficient design of file systems, database systems, and applications built on top of them. For instance, the upper layer can remove some write barriers and data replication to achieve higher performance.

In this paper, we introduce the *snapshot-consistent flash translation layer* (SCFTL). The *flash translation layer* (FTL) is the interface between flash memory and upper layers in the storage stack, providing operations such as write, read, and flush. As the name suggests, SCFTL implements the snapshot-consistent disk model, which ensures that a crashed disk will recover to the state right before the last flush operation.

Our snapshot-consistent disk model has the benefit that the flush operation can be used to take a "disk snapshot." The feature is particularly useful for upper layers to implement atomic operations/transactions—they only need to invoke a flush at the end of each operation/transaction. Upper-layer systems can utilize this feature to obtain a more efficient design, e.g., removing the journal from a file system.

Next, we compare the snapshot-consistent disk model with other disk models used in the literature. At first glance, one might think that the *synchronous disk model* can provide a similar crash guarantee, as it also confines the number of post-crash states to one. The synchronous disk model, however, can ensure only the atomicity of a single disk write, whereas our proposed snapshot-consistent disk model guarantees the atomicity of multiple writes between two consecutive flushes.

The asynchronous disk model guarantees only that writes before a flush are durable. For those after the last flush operation, even the order is not guaranteed. In the worst case,

it might have 2^n post-crash states, where n is the number of writes after the last flush. The *prefix-preserving disk model* guarantees that writes after the last flush will not be reordered, so the possible post-crash states reduce to, in the worst case, n. The post-crash states of the two disk models are "non-deterministic," and the upper layers have to consider all possible scenarios in their crash recovery mechanisms.

Our SCFTL allows an efficient implementation (§3) utilizing the *out-of-place* update feature of FTLs. An FTL usually maintains an in-memory *logical-to-physical* address translation table (or L2P for short). On a write of data d to a logical address l, due to the physical constraint of the flash memory, the FTL cannot just update the value pointed to by l to d. Instead, it finds a new physical location, puts d there, and updates L2P to remap l to that new location. The old data pointed to by l remains there. The main idea of our implementation is to remember the L2P right before the last flush operation, which we refer to as the *stable L2P*. When the system crashes, we use the stable L2P to recover to the state before the last flush. Writing the entire L2P to the flash memory is an expensive operation, so we design a mechanism to store only the changes to the last stored L2P table, and store the full L2P to the flash memory only occasionally. We also designed a mechanism to ensure that the garbage collector will not erase the data pointed by the stable L2P.

We have formally verified the correctness of our SCFTL implementation using a combination of a proof assistant, a symbolic executor, and an SMT solver, which achieves a good balance between the degree of automation and the expressive power for stating and proving desired properties. The formal framework is set up manually using an interactive proof assistant, while the proof obligations involving the detail of SCFTL are discharged automatically with an SMT solver.

In the formal framework, we start with a simple mathematical specification of the snapshot-consistent disk model, which we briefly illustrate here using Figure 1. The state of the specification has two sector arrays, **stable** and **volatile**. The flush() operation copies **volatile** to **stable**; then the operation write$(0, x_9)$ changes **volatile**$[0]$ to x_9 and nothing else, while the read(2) operation returns **volatile**$[2]$; finally, the recovery() operation overwrites **volatile** with **stable**. In contrast to the specification, the SCFTL implementation stores the two arrays using more sophisticated data structures to achieve better performance and to satisfy the constraints of the flash memory. For instance, the **stable** array is implemented as an in-flash L2P and a list of L2P changes. Using a proof assistant, we reduce a *behavioral correctness* property over multiple FTL operations—which asserts that the SCFTL implementation behaves as the specification describes—to simpler *per-operation correctness* properties about each FTL operation (§4). We also prove that the behavioral correctness of SCFTL implies its snapshot consistency.

We then use more *automatic* tools to prove per-operation correctness (§5). More specifically, the relationship between

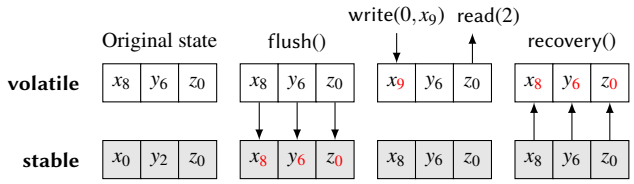


Figure 1. Illustration of the SCFTL operations.

the two arrays in the specification and the data structures in our implementation is described as a logical formula called the *abstraction relation*, and one type of the per-operation correctness formulae has the form "if the abstraction relation holds for the states before executing an operation, the relation will remain true for the states after executing the operation." We use a *symbolic executor* [8, 32] to translate the C program of our SCFTL implementation to logical formulae describing how the states change after executing an SCFTL operation. Then we use an SMT solver to ensure that our implementation does satisfy the per-operation correctness formulae.

Our experimental results (§7) show that when a workload does not flush the disk too frequently, SCFTL is as efficient as an FTL implementing the asynchronous disk model. To understand the usefulness of SCFTL, we modify the xv6 [16] file system to support group commit and utilize the stronger crash guarantee granted by SCFTL. By changing less than 30 lines of code, we show that the modified xv6 on SCFTL outperforms xv6 with logging on conventional FTLs by 3 to 30 times using our file system benchmarks; the performance improvement is less obvious for workloads that frequently flush the disk (e.g., smallfiles repeatedly creates a file, writes 100 bytes of data to it, and calls `fsync`), and more obvious for workloads with lower flush frequency (e.g., largefile writes 4 MB of data to a file and calls `fsync` for every 1 MB). This observation suggests an important guideline for building systems and applications on top of SCFTL: reducing the flush frequency to extract more benefits from SCFTL. Finally, we use the same file system benchmarks to compare the performance of the modified xv6 on SCFTL with the state-of-the-art setting: the ext4 file system on the pblk [6] FTL. The result is encouraging. Although xv6 is a file system known to be simple but slow, our xv6 on SCFTL is in the worst case only two times slower than ext4 on pblk. Moreover, xv6 on SCFTL has a stronger crash guarantee than that of ext4 on pblk.

In summary, the main contributions of this paper are the design, specification, and verification of SCFTL:

- The design exploits the out-of-place update feature of FTLs and uses an efficient checkpointing algorithm to provide a stronger crash guarantee at the disk level (§3). We validate its efficiency with disk and file system benchmarks (§7).
- The specification is simple and useful as it involves only the manipulation of two arrays and a counter, and it naturally ensures the atomicity of multiple disk writes. We formalize snapshot consistency and show that the specification satisfies the property with a proof assistant (§4).

- We verify that our implementation of SCFTL meets its specification using automatic verification tools. To scale the verification of SCFTL, we propose a novel approach to model crash behavior and describe some techniques to simplify the proof obligations, including using ghost variables to craft efficient SMT encodings, categorizing invariants to remove unnecessary conditions, and partitioning the proofs to avoid non-determinism (§5).

SCFTL has several limitations. First, SCFTL does not allow concurrent SCFTL operations, although it does allow flash operations to be executed concurrently (more detail in §6.2). Second, SCFTL assumes the underlying flash memory is free of error. Finally, SCFTL does not implement standard optimizations of FTLs, such as hot-cold data separation and wear leveling, although its design does not prohibit them.

2 Related Work

We first discuss the recent advance of disk models; in particular, we will focus on *transactional* and *order-preserving* models. To the best of our knowledge, our work is the first to address the verification of the crash safety issue at the physical device layer. Most previous work on crash safety verification assumes a correct asynchronous disk is given and put their focus on other layers in the storage stack, e.g., the file system. We will discuss some recent work in this direction.

The *transactional models* [15, 17, 25, 38, 41] guarantee the atomicity of multiple write operations. They provide a non-standard disk interface, which has two consequences: (i) their semantics (e.g., isolated concurrent transactions and transaction abortion) is hard to formally specify or verify, and (ii) system developers have to learn a new interface, increasing the burden of porting existing or developing new software.

The *order-preserving disk models* [11, 45] guarantee the preservation of operation orders across a crash. They expose the standard read-write-flush disk interface, but with fewer possible post-crash states than the asynchronous disk model. Upper layers in the storage stack can utilize this feature to reduce the number of flushes invoked in their crash recovery mechanisms (e.g., copy-on-write and journaling).

Compared with the transactional models, the snapshot-consistent model uses the standard read-write-flush interface. It, moreover, guarantees the atomicity of multiple writes between two consecutive flushes and thus provides a stronger guarantee than the order-preserving models.

Recent research work has discovered many crash vulnerabilities in widely used applications such as LevelDB and Git [36], as well as ACID violations in many relational database systems [46].These vulnerabilities mainly stem from the vague and weak crash guarantees provided by the underlying file systems. File systems themselves, even for mature ones such as ext4, btrfs [39] and F2FS [28], also contain bugs that may result in severe consequences [24, 27, 31].

In the past decades, a significant amount of research effort is devoted to the development of a verified crash-safe storage stack. To name a few, the verified file systems Yxv6 [42], FSCQ [13], and DFSCQ [12] assume an asynchronous disk model and use a log-based design to guarantee crash safety. Instead of the asynchronous disk model, both the BilbyFS [3] and Flashix [19] file systems assume the underlying layer is a raw flash device (without an FTL) and implement an atomic transaction mechanism to ensure crash safety.

SCFTL differs from previous work on verifying the storage stack in that it targets the physical device layer. This approach has three notable benefits: First, the code and the data structure of an FTL are usually simpler and more manageable than that of a file system; thus it is easier to develop an efficient yet verifiable layer. Second, providing the standard disk interface is more modular than directly building file systems on a raw flash device; developers can build their own systems with various features and optimizations, and leave crash safety to the underlying verified disk. Finally, this approach allows us to exploit useful device characteristics (e.g., the out-of-place update feature of FTLs); it enables SCFTL to provide a stronger crash guarantee without compromising the performance.

Regarding the verification methodology and framework, the closest work to ours is Yggdrasil [42], which establishes a forward simulation and discharges the proof obligations using an SMT solver, achieving a high degree of automation. Compared with Yggdrasil, we have additionally formalized our simulation argument, snapshot consistency, and relevant theorems using a proof assistant, providing more correctness guarantees. The proof obligations have to be manually massaged into a form that can be handled by an SMT solver; strictly speaking, this leaves a gap in our formal proof (but we bridge the gap by pen-and-paper reasoning).

By contrast, there has been work on storage system verification [9, 10, 13, 19] where the entire proof is formally verified with a proof assistant and does not have any gaps, although their verification cost is also significantly higher since the whole proof structure has to be carefully designed and constructed by programmers. Our framework is not as sophisticated as Argosy [9], which treats layered storage systems, or Perennial [10], which supports concurrency; extending our framework to support these features are interesting future directions. There are also differences in modeling decisions between our framework and the others—for example, in Argosy a crash is modeled as an individual event whereas we incorporate crashes into operations, and in Yggdrasil the effect of a lower-level operation may not be visible at a higher level whereas we always relate such an operation to a corresponding higher-level operation that is specified not to change the state. These differences in modeling decisions do not lead to vital differences in correctness guarantees, however.

3 SCFTL Design and Implementation

SCFTL is designed with *high performance*, *strong crash guarantees*, and *provable correctness* in mind. Below we give an

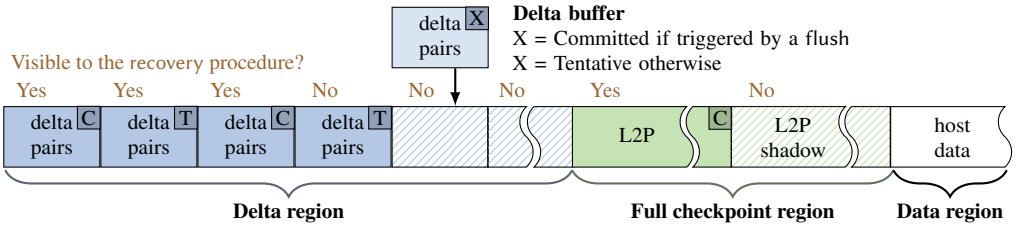


Figure 2. Flash memory layout of SCFTL.

overview of flash disks and describe the techniques we use that set SCFTL apart from traditional FTLs.

3.1 Flash disk overview

Usually, a flash disk contains two main components, a fast *dynamic random-access memory* (DRAM) to store temporary data and a slower *flash memory* to store permanent data. In this paper, we use the keywords *in-memory* or *buffer* to mean the data is stored in the DRAM and *in-flash* to mean it is stored in the flash memory. Flash memory is usually structured as a list of *blocks*, each of which consists of multiple *pages*. A page further contains one or multiple *sectors*, the basic unit the upper layers (e.g., the file system) use to access data. One can access flash memory through commands such as READ and WRITE a page, ERASE a block, and SYNC to wait for the completion of all ongoing flash commands. Due to physical limitations, flash commands need to follow several intricate constraints. For instance, a *page* must be erased before being written. However, the basic unit for ERASE is a block, while that for WRITE is a page. It would be inefficient if we erase the entire block whenever we write to a page within it.

In order to free users from handling the intricate device characteristics of flash memory, a flash disk usually comes with a flash translation layer (FTL) to hide the complexity. Typical operations supported by an FTL include a write and a read operation to store and retrieve a sector of data, a flush operation to wait until all unprocessed changes is made to the flash memory, a recovery procedure that will be invoked after a crash, and a garbage collection (GC) procedure gc that will only be invoked by its internal garbage collector. Every FTL should at least support the main functionalities, namely *address translation*, *crash recovery*, and *garbage collection*. Below we introduce how SCFTL implements those functions.

3.2 Address translation

To comply with the *erase-before-write constraint* of flash memory, SCFTL maintains an in-memory logical-to-physical table (L2P) and writes data in a log-structured manner [40] to avoid in-place updates. Address translation can be done at the granularity of sectors, pages, blocks, or a mixture of them [26]. Often finer granularity leads to better performance, but at the cost of higher memory usage due to a larger L2P. SCFTL uses a sector-level L2P to achieve better performance.

SCFTL handles the request write(la, d), i.e., writing a sector of data d to the logical sector address la, as follows. It first stores d into a page-sized *merge buffer* employed to resolve the size mismatch between a sector and a page. Then SCFTL finds a new location pa for placing d using the triplet (blk, pg, sec), called an *active pointer*, where the first two together point to the next free page to be written, and the last one points to the next free slot in the merge buffer. We call the block blk the *active block*. Then SCFTL also updates the in-memory L2P with the entry $la \mapsto pa$.

When the merge buffer is full, SCFTL invokes the command WRITE(blk, pg, d) to write the buffered data to the flash memory, where d is the content of the merge buffer. Then SCFTL increases pg by one to follow the *sequential write constraint* (within one block) of flash memory, unless pg is already the last page in a block, in which case blk is assigned a new block address dequeued from the *free block queue* and pg is reset to 0. The free block queue is an in-memory data structure that SCFTL uses to track currently available blocks.

Finally, SCFTL would have to remember that the address storing old data (if any) is no longer valid, and the one for the new data is now valid. The garbage collector needs this information to relocate all valid data before erasing a block. In SCFTL, this is realized by an in-memory physical-to-logical table (P2L), which is the "reverse" mapping of L2P. Flash disks usually have more available physical locations than logical locations. So it can happen that a mapping $p \mapsto l$ is in P2L, but $l \mapsto p$ is not in L2P. In such a case, we know that this physical address p is invalid. We also use an in-memory table, called the *valid count table*, to remember the number of valid sectors in each block. The valid count table will be used by the garbage collector to select the *victim block* to recycle.

Handling a read(la) request is simpler: SCFTL first checks if the requested data is still stored in the merge buffer; if so, it directly returns the data in the merge buffer. Otherwise, SCFTL performs an L2P lookup to find the corresponding physical address p, followed by a READ(p) command to retrieve the requested data stored in the flash memory. For a flush() request, SCFTL first stores the buffered data in the flash memory and advances the active pointer in the same way as handling a write request. Then it issues a SYNC command to ensure all data written before this point is persistent.

SCFTL maintains another L2P (the *stable L2P*) to reflect the state of the in-memory L2P right before the last flush for ensuring snapshot consistency. To distinguish the two kinds of L2P, we will call the in-memory L2P the *volatile L2P*. The cost of physically storing the stable L2P in the flash memory can be prohibitively high. Thus, SCFTL logically maintains the stable L2P through *checkpointing* in an efficient way.

Figure 2 shows the flash memory layout of SCFTL, including a *delta region* that records L2P differences, and a *full*

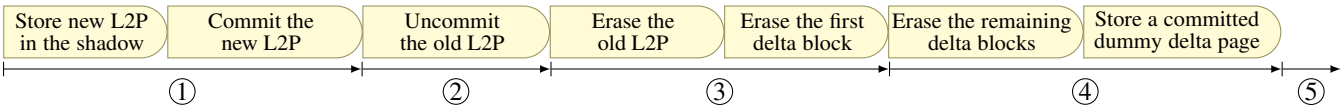


Figure 3. Full checkpoint protocol. The occurrence of crashes is partitioned to stages ① - ⑤.

checkpoint region that stores an entire L2P. SCFTL can create two kinds of checkpoints: a lightweight *delta checkpoint* and a heavyweight *full checkpoint*. SCFTL always creates a delta checkpoint when a flush is invoked. Under rare circumstances where the delta region nearly runs out of space, SCFTL creates a full checkpoint and clears the delta region.

Delta checkpoint Whenever a volatile L2P entry is modified due to a write or a gc operation, SCFTL also inserts a *delta pair*, which consists of a logical and a physical address, into the page-sized *delta buffer*. When the delta buffer is full, SCFTL invokes a WRITE command to store all buffered pairs along with a *tentative tag* into the in-flash delta region in a sequential manner and clears the buffer. We illustrate this operation in Figure 2.

On receiving a flush request, SCFTL first makes sure host data is safely stored in the flash memory. It then follows the same procedure above to store the buffered pairs in the flash memory, except here the delta page is tagged as *committed*. A committed delta page *activates* previous tentative delta pages, i.e., all delta pages before a committed one are treated as committed. When the committed delta page is safely kept in the flash memory, a delta checkpoint is successfully created.

Full checkpoint A full checkpoint of SCFTL consists of a complete L2P table and a commit flag. When making a full checkpoint, i.e., storing the volatile L2P together with a commit flag to the flash memory, we employ a *shadow* to prevent modification to the old L2P before the new one is settled. The detailed steps can be found in Figure 3. In short, we first store the new (volatile) L2P to the flash memory and start to erase the old L2P and the delta region only after the new L2P is committed. To ensure the correctness of our recovery procedure, we allow only flush operations to create a full checkpoint; write and gc operations are not allowed to create a full checkpoint. One potential issue is that the delta region might become full after a write or a gc operation. We address this issue by imposing upper bounds on the number of write and gc invocations (hence the number of created delta pairs) within an *epoch*, i.e., between two consecutive flushes. To ensure these bounds are respected, SCFTL uses a *write counter* and a *GC counter* to keep track of the number of write and gc invocations in the current epoch, respectively, and resets both counters on a flush or a recovery. If the upper layer calls a write after the write counter exceeds the bound, SCFTL simply treats that write as a no-op.

Systems that use SCFTL to implement atomic transactions should be aware of the write bound, to make sure an operation can fit into the current epoch before executing it. If an operation is too large to fit into an entire epoch (e.g., writing a large amount of data to a file), then it should be broken into multiple smaller ones. We believe that this requirement is not too restrictive, given that some systems also face a similar situation; for example, because of the log size limit, ext4 always checks that the current running transaction has sufficient capacity left before atomically updating its metadata [35, §3.1]. Similarly, systems built on SCFTL can record the number of writes issued to SCFTL since the last flush, and *before* executing an operation, calculate the number of writes required to complete the operation. If the operation does not fit into the current epoch, then the system should call a flush to form a new epoch. This avoids calling a flush in the middle of an operation, which may expose intermediate states on a crash.

3.3 Crash recovery

The recovery procedure first recovers the volatile L2P with full and delta checkpoints. To explain how to reconstruct the L2P, we begin by specifying what is *visible* to the recovery procedure. A full checkpoint is visible if and only if it is committed. For the delta region, we treat its first page specially to ensure we can erase the entire delta region atomically. The only important information on the first page is the commit flag. All delta pairs on the first page are *dummy delta pairs* that will be ignored in the recovery procedure. Visibility of delta pairs can be determined by examining the commit flag of the first page of the delta region and, if the commit flag is on, performing a sequential scan over the entire delta region to find the last committed page.

In most cases, the recovery procedure simply restores a base L2P from a full checkpoint and applies all visible delta pairs in sequential order. The only exception is when a crash occurs during a full checkpoint. Below we analyze the behavior of the recovery procedure against each crash point during a full checkpoint, as shown in Figure 3:

①: The old L2P and all delta pairs are visible. Restoring the old L2P and applying each delta pair yields the new L2P.

②: Both the old and new L2P, and all delta pairs are visible. Although we do not leave other information (e.g., a timestamp) to distinguish the old L2P from the new one, our checkpoint design ensures that starting from either L2P and applying each delta pair restores the new L2P.

③: The new L2P and all delta pairs are visible. For the same reasoning in ②, it restores the new L2P and applies each delta pair, yielding the new L2P.

④: The new L2P is visible and all delta pairs are invisible. It simply restores the new L2P.

⑤: The new L2P and the dummy pairs are visible. It restores the new L2P and ignores all dummy delta pairs.

Selective persistence The idea of selective persistence [34] is to persistently keep only *primary data*, which is the minimal set of data structures required for correct crash recovery,

and rebuild non-primary data from the primary one. This way, the protocol to correctly maintain consistency between multiple data structures can be greatly simplified. In SCFTL, the primary data is simply the L2P. After restoring the volatile L2P, the recovery procedure proceeds to rebuild the *block queues* (explained later), P2L table, and valid counts table.

3.4 Garbage collection

The GC procedure of standard FTLs consists of the following steps: (i) find a *victim block*, (ii) relocate all valid sectors (those pointed to by the volatile L2P) in the victim block similarly to the write operation (the *relocation phase*), and (iii) erase the victim block (the *erasure phase*). In SCFTL, the standard GC procedure cannot be used, as the data pointed to by the stable L2P might be erased by a garbage collector.

To prevent such an issue, one design option is to keep additional information on what is allowed to be erased. This approach is taken by TxFlash [38, §3.3] to prevent back pointers from being erased. The downside of this approach is that it can incur notable memory and performance overhead.

Another approach, adopted by OPTR [11, §3.4], is to invoke an internal flush (i.e., a flush operation issued by the FTL) before GC is activated. While an internal flush is allowed by the prefix-preserving guarantee that OPTR offers, it is not allowed by the snapshot consistency SCFTL is trying to achieve, as otherwise SCFTL might rollback to the state right before an internal flush on a recovery.

Two-phase garbage collection Instead, we use a simple protocol called *two-phase garbage collection* (2PGC), which delays the erasure phase until a flush is invoked. This is correct because after a flush operation, the old stable L2P will be discarded and hence all the previously selected victim blocks can be safely erased. To implement this idea, in the 2PGC mechanism, we use two functions $\mathrm{gc}_{\mathsf{rl}}$ and $\mathrm{gc}_{\mathsf{es}}$ to handle the relocation and erasure phases, respectively. We, moreover, maintain four queues to remember the status of blocks. Initially, all blocks are in the *free block queue*, except the active block, which does not belong to any of the state queues. When the active block is fully written, we add it to the *used block queue* and pick another block from the *free block queue* as the new active block.

When the garbage collector invokes the $\mathrm{gc}_{\mathsf{rl}}$ function, it first removes a victim block from the used queue and performs the relocation of valid sectors. After all valid sectors are removed from the victim block, the $\mathrm{gc}_{\mathsf{rl}}$ function adds the victim block to the *invalid block queue*. As the name suggests, all blocks in the invalid queue do not have any valid sector. Nevertheless, those blocks might still contain sectors pointed to by the stable L2P and hence cannot be immediately erased. All blocks in the *invalid* queue will be moved to the *erasable block queue* at the end of a flush operation. After a flush, the stable L2P will be updated, and all invalid blocks are no longer pointed to by the new stable L2P and are now erasable. All blocks in the erasable block queue can be safely erased. When the garbage collector invokes the $\mathrm{gc}_{\mathsf{es}}$ function, it finds a block b in the erasable queue, performs an ERASE(b) command, and puts b in the free block queue.

The remaining problem of 2PGC is that garbage collected blocks cannot be immediately reused in the current epoch. To address this issue, SCFTL exploits the upper bounds on the number of write and $\mathrm{gc}_{\mathsf{rl}}$ allowed in one epoch to ensure that there is sufficient space for newly written data and relocated data. These two bounds already exist to avoid overflowing the delta region (§3.2). Below we describe the constraints that need to be satisfied when picking the values for the two bounds, W (for write, in terms of sectors) and K (for $\mathrm{gc}_{\mathsf{rl}}$, in terms of blocks).

The first constraint ensures that write and $\mathrm{gc}_{\mathsf{rl}}$ do not consume space more than what $\mathrm{gc}_{\mathsf{rl}}$ can produce in one epoch:

$$\underbrace{W}_{\text{consumed by write}} + \underbrace{KN}_{\text{consumed by gc}_{\mathsf{rl}}} \leq \underbrace{KS}_{\text{produced by gc}_{\mathsf{rl}}} \quad (1)$$

where N is the maximum number of valid sectors in every victim block (we will explain how to obtain this bound later); S is the number of sectors per block. Each write occupies one sector and each $\mathrm{gc}_{\mathsf{rl}}$ relocates at most N sectors; thus at most $W+KN$ sectors will be consumed in one epoch. Each $\mathrm{gc}_{\mathsf{rl}}$ also turns one used block into one invalid block, which becomes an erasable block in the next epoch; thus at most KS sectors can be produced in one epoch.

To obtain the bound N, we introduce a GC threshold U: $\mathrm{gc}_{\mathsf{rl}}$ can be activated only when the number of used blocks is larger than or equal to U. Let L be the number of entries of L2P (i.e., the number of logical sectors). Recall that a valid sector is a physical sector mapped by the volatile L2P. The *pigeonhole principle* states that there exists a used block (i.e., the hole) whose number of valid sectors (i.e., the pigeons) is no more than $\lfloor L/U \rfloor$. One design choice is to force $\mathrm{gc}_{\mathsf{rl}}$ to always pick the block with the least number of valid sectors as the victim block (i.e., the greedy policy). A more flexible alternative allows $\mathrm{gc}_{\mathsf{rl}}$ to pick any block providing the block has no more than $\lfloor L/U \rfloor$ valid sectors; the pigeonhole principle ensures the existence of such block. Either way, we obtain $N = \lfloor L/U \rfloor$.

The threshold may disable $\mathrm{gc}_{\mathsf{rl}}$ before there is enough space for the next epoch; SCFTL would fail to proceed to the next epoch in this situation. To avoid such a situation, we can choose a proper U, W, and K such that $\mathrm{gc}_{\mathsf{rl}}$ must be enabled when there is not enough space for the next epoch:

$$\underbrace{U}_{\text{GC threshold}} \leq \underbrace{P - 1 - \lceil (W+KN)/S \rceil}_{\text{lower bound of used blocks when not enough space}} \quad (2)$$

where P is the number of data blocks. First observe that free, erasable, and invalid blocks are all available to the next epoch as all invalid blocks become erasable after a flush. The condition of "not enough space" essentially means that the total number of these three kinds of blocks is less than $\lceil (W+KN)/S \rceil$. Given that the total number of free, erasable, invalid, and used blocks is equal to $P-1$ (we always have

one active block), we know that the number of used blocks is more than $P-1-\lceil (W+KN)/S \rceil$ when there is not enough space for the next epoch.

We demonstrate a configuration that satisfies the above constraints: A 4-GB flash disk has 2^{20} logical sectors ($L = 2^{20}$); suppose each block has 512 sectors ($S = 512$) and we have 6 GB of flash memory for storing data, then the number of data blocks will be 3072 ($P = 3072$). We can set the GC threshold at 2500 ($U = 2500$) and we know the number of valid sectors in every victim block would not exceed 419 ($\lfloor L/U \rfloor = \lfloor 2^{20}/2500 \rfloor = 419$). Next we pick suitable values for W and K. Let $W = 4000$ and $K = 50$ and check whether they satisfy the above two constraints:

$$24950 = W + KN \leq KS = 25600$$
$$2500 = U \leq P - 1 - \lceil (W+KN)/S \rceil = 3022$$

Both constraints are satisfied.

Recovery of block queues After the recovery procedure reconstructs the volatile L2P, it proceeds to reconstruct other data structures, including the aforementioned block queues. The recovery procedure simply scans through the blocks. If a block contains some sectors pointed to by the L2P, then it is a used block; otherwise, the recovery procedure treats it as an erasable block. Note that after the reconstruction of the volatile L2P, the stable and volatile L2Ps are identical and hence we do not have any invalid blocks. We do not have any free blocks after the recovery procedure; we do not have the information whether a block was in the erasable or free block queue before the crash. So we have to play it safe and put them in the erasable queue to follow the erase-before-write flash constraint.

4 Formal Verification Framework

The design of SCFTL (§3) is fairly sophisticated; to provide a strong guarantee of its reliability, we have formally verified its correctness. Our formal verification framework starts with the definition of a disk model $\mathbb{S}$ (similar to Figure 1) that specifies the intended disk behavior. $\mathbb{S}$ is defined as a particular kind of state transition system (§4.1) on which snapshot consistency can be formulated and proven (§4.2). As opposed to $\mathbb{S}$, which is merely an abstract, mathematical specification that is meant to be understood easily, the SCFTL implementation constitutes a more realistic transition system $\mathbb{P}$. We prove that $\mathbb{P}$ is behaviorally correct with respect to $\mathbb{S}$, and moreover, this behavioral correctness is strong enough to imply the snapshot consistency of $\mathbb{P}$ (§4.3). Behavioral correctness is a complicated property about sequences of state transitions, which cannot be easily verified with automatic verification tools. We can, however, reduce its proof to one about the behavior of individual operations (§4.4), the latter of which is more amenable to automatic verification (§5). The content of this section is formally verified with the Agda proof assistant [33], but here we provide only a high-level sketch.

4.1 Specification of disk behavior

To model the behavior of a disk as a state transition system, we should define the possible states of a disk and the operations that can be performed on the disk states. In $\mathbb{S}$, which is our abstract disk model that acts as a definition of intended behavior, a state t of a disk is a pair of arrays $t.volatile$ and $t.stable$ representing the volatile and stable copies of disk data respectively and a number $t.wcnt$ that counts the number of writes since the last flush. There is a set $\mathcal{N}_{\mathbb{S}}$ of states that represent the possible contents of a new disk, where only the *stable* array is initialized to some default value. Reading a disk state is just retrieving the data at a given address in the volatile part; since the operation does not change the disk state, we simply define it as a function $read(t,a) \triangleq t.volatile[a]$ rather than a kind of state transition.

Mirroring the FTL operations except read, the operations of $\mathbb{S}$ are shown in Figure 4: they are classified as *regular*, *flush*, and *recovery* operations, and have a successfully executed version and a crashed version. Writing $t \xrightarrow{op} t'$ to mean that there is a transition from t to t' through the operation op, that is, t' is the state resulting from applying the operation op to the state t, we define the effect of an operation by specifying how t and t' are related: A write operation $\mathsf{w}_{a,d}$, which writes the piece of data d to the address a in the volatile part, is defined by saying that $t \xrightarrow{\mathsf{w}_{a,d}} t'$ amounts to

- $t'.volatile = t.volatile[a \mapsto d]$, where the right-hand side is the array whose values are the same as $t.volatile$ except at the address a, where the value is d,
- $t'.stable = t.stable$, meaning that the stable data is not modified, and
- $t'.wcnt = t.wcnt + 1$, meaning that $wcnt$, the write counter mentioned at the end of §3.2, is incremented by one

when a and $t.wcnt$ are within bounds, or otherwise $t' = t$. The garbage-collecting operations rl and es do not change the (abstract) disk state. The flush operation f copies the volatile data to the stable part, and the recovery operation r does the opposite; both operations reset the write counter to 0. The crashed operations $\mathsf{w}^{\mathsf{c}}_{a,d}$, rl^{c}, es^{c}, and r^{c} may disrupt the volatile data arbitrarily, and thus their definitions only specify that the stable data remains the same (and the transitions become non-deterministic); for f^{c} there are two kinds of post-crash state because the update to the flash disk may have finished, in which case the system behaves as if the flush operation is successfully executed.

4.2 Snapshot consistency

Snapshot consistency is essentially a property about *execution fragments* (or *fragments* for short), which are consecutive sequences of transitions $t_1 \xrightarrow{op_1} t_2 \xrightarrow{op_2} \cdots \xrightarrow{op_n} t_{n+1}$; we often omit unimportant intermediate states and write $t_1 \xrightarrow{op_1, op_2, \ldots, op_n} t_{n+1}$. Informally, snapshot consistency says that when recovered from a crash, reading the state after the recovery operation will be the same as reading the state right

	Regular		Flush	Recovery
	Write	**Relocate/Erase (GC)**	**Flush**	**Recover**
Successful	$t \xrightarrow{\mathsf{w}_{a,d}} t' \triangleq$ $(\mathit{InBounds}(a, t.wcnt) \wedge t'.volatile = t.volatile[a \mapsto d]$ $\wedge\, t'.stable = t.stable \wedge t'.wcnt = t.wcnt + 1)$ $\vee\, (\neg \mathit{InBounds}(a, t.wcnt) \wedge t' = t)$	$t \xrightarrow{\mathsf{rl/es}} t' \triangleq$ $t' = t$	$t \xrightarrow{\mathsf{f}} t' \triangleq$ $t'.volatile = t.volatile$ $\wedge\, t'.stable = t.volatile$ $\wedge\, t'.wcnt = 0$	$t \xrightarrow{\mathsf{r}} t' \triangleq$ $t'.volatile = t.stable$ $\wedge\, t'.stable = t.stable$ $\wedge\, t'.wcnt = 0$
Crashed	$t \xrightarrow{\mathsf{w}^{\mathsf{c}}_{a,d}} t' \triangleq$ $t'.stable = t.stable$	$t \xrightarrow{\mathsf{rl}^{\mathsf{c}}/\mathsf{es}^{\mathsf{c}}} t' \triangleq$ $t'.stable = t.stable$	$t \xrightarrow{\mathsf{f}^{\mathsf{c}}} t' \triangleq$ $t'.stable = t.volatile$ $\vee\, t'.stable = t.stable$	$t \xrightarrow{\mathsf{r}^{\mathsf{c}}} t' \triangleq$ $t'.stable = t.stable$

Figure 4. Definitions of operations in $\mathbb{S}$. GC stands for Garbage Collection. The definition of the predicate $\mathit{InBounds}(a, wcnt)$ is $a \le a_{max} \wedge wcnt \le wcnt_{max}$ where a_{max} and $wcnt_{max}$ are the upper bounds on the addresses and the write counter respectively. Note that in the definitions a write operation has no effect ($t' = t$) when the write counter exceeds the bound ($wcnt > wcnt_{max}$).

before the last flush operation prior to the crash. More precisely, the disk may have operated normally for some time before the crash happens, and the recovery may fail several times before it succeeds. This whole behavior is described as a *one-recovery fragment* of the form

$$t_1 \xrightarrow{a_1,\ldots,a_{k-1}} t_2 \xrightarrow{a_k(=\mathsf{f}),b_1,\ldots,b_\ell} t_3 \xrightarrow{c,(\mathsf{r}^{\mathsf{c}})^m,\mathsf{r}} t_4$$

where $a_1, \ldots, a_k$ are a sequence of successful regular or flush operations ending with f (that is, $a_k = \mathsf{f}$), $b_1, \ldots, b_\ell$ are successful regular operations, c is a crashed regular or flush operation, and $(\mathsf{r}^{\mathsf{c}})^m$ is the crashed recovery operation repeated m times; this is abbreviated to $t_1 \rightsquigarrow t_4$ later on. Writing $t \approx t'$ to mean that $\mathit{read}(t, a) = \mathit{read}(t', a)$ for all addresses a within bounds, snapshot consistency of a one-recovery fragment of the above form is defined as follows:

- if c is a crashed regular operation, then $t_2 \approx t_4$;
- if c is the crashed flush operation f^{c}, then either $t_2 \approx t_4$ or $t_3 \approx t_4$.

Note that in the definition k can be 0, in which case no flush is performed before the crash c, and the definition requires, for instance in the first case, that the disk be reverted to the first state t_1 (which equals t_2) observationally.

We can now formulate snapshot consistency of a disk model. The typical way of using a disk can be represented as a *multi-recovery fragment* of the form

$$t_0 \xrightarrow{(\mathsf{r}^{\mathsf{c}})^\ell,\mathsf{r}} t_1 \rightsquigarrow t_2 \rightsquigarrow \cdots \rightsquigarrow t_n \rightsquigarrow t_{n+1} \xrightarrow{a_1,\ldots,a_m} t_{n+2}$$

which starts with performing the recovery operation on a state $t_0 \in \mathcal{N}_{\mathbb{S}}$ (until it succeeds) to bring the disk to a usable state, and continues with an arbitrary number of one-recovery fragments and some trailing regular and flush transitions representing uses of the disk. We say that a multi-recovery fragment of this form is snapshot-consistent if all the one-recovery sub-fragments $t_1 \rightsquigarrow t_2, \ldots, t_n \rightsquigarrow t_{n+1}$ are snapshot-consistent, and that a disk model is snapshot-consistent if its every multi-recovery fragment is snapshot-consistent. With the definitions in place, we can now state our first result (whose proof is straightforward and omitted here).

Lemma 1. *$\mathbb{S}$ is snapshot-consistent.*

Note that the definitions of snapshot consistency make sense for any disk model that has the same structure as $\mathbb{S}$, in particular for the model $\mathbb{P}$ that we will describe next.

4.3 Behavioral correctness and snapshot consistency of SCFTL

$\mathbb{S}$ is a simplistic transition system: it gives a concise definition of the intended disk behavior, but is unsuitable for direct implementation. In SCFTL (§3), we use more practical states that consist of various in-memory and in-flash data structures, and sophisticated operations implemented in C. All these give rise to another transition system $\mathbb{P}$, which has the same set of operations as $\mathbb{S}$ (as well as a *read* function for reading the states of $\mathbb{P}$) and also a set $\mathcal{N}_{\mathbb{P}}$ of possible states of a new disk (where only the in-flash part is initialized). The definition of $\mathbb{P}$ is a formal version of what has been presented in §3; the exact definition is not needed in this section though, and will be described later in §5.

We have proven that $\mathbb{P}$ is *behaviorally correct* with respect to $\mathbb{S}$: if we perform a legitimate sequence of operations in $\mathbb{P}$ to obtain a fragment and read the *normal* states, which are states that immediately follow a successful operation, the results will be the same as performing the same sequence of operations in $\mathbb{S}$ and reading the corresponding states. This property allows the behavior of $\mathbb{P}$ to be understood in terms of $\mathbb{S}$. More formally, we have the following theorem.

Theorem 1 (behavioral correctness of $\mathbb{P}$)**.** *For every multi-recovery fragment $s_0 \xrightarrow{(\mathsf{r}^{\mathsf{c}})^m,\mathsf{r}} s_1 \xrightarrow{op_1} \ldots \xrightarrow{op_n} s_{n+1}$ in $\mathbb{P}$, there exists a fragment $t_0 \xrightarrow{(\mathsf{r}^{\mathsf{c}})^m,\mathsf{r}} t_1 \xrightarrow{op_1} \ldots \xrightarrow{op_n} t_{n+1}$ in $\mathbb{S}$ (which has the same sequence of operations) such that $s_i \approx t_i$ for every corresponding pair of normal states s_i and t_i.*

We will see how Theorem 1 is proven in §4.4 and §5. Before we do so, we show that the behavioral correctness of $\mathbb{P}$ is strong enough to allow $\mathbb{P}$ to inherit snapshot consistency from $\mathbb{S}$.

Theorem 2. *$\mathbb{P}$ is snapshot-consistent.*

Proof. We must show that any multi-recovery fragment in $\mathbb{P}$ is snapshot-consistent, that is, the results of reading the states mentioned by the definitions of snapshot consistency are the same. Observe that all these states are normal, so reading these states in the fragment (in $\mathbb{P}$) is the same as reading the

corresponding states in the fragment in $\mathbb{S}$ that is guaranteed to exist by the behavioral correctness of $\mathbb{P}$. This means that the $\mathbb{P}$-fragment is snapshot-consistent if the $\mathbb{S}$-fragment is, and the latter is indeed the case because $\mathbb{S}$ is snapshot-consistent (Lemma 1). □

4.4 Per-operation correctness

Theorem 1 is proven with a *forward simulation* argument [30], which, though fairly standard, is described below for the sake of completeness. Given a multi-recovery fragment in $\mathbb{P}$, we construct a fragment in $\mathbb{S}$ with the same sequence of operations while ensuring that every corresponding pair of normal states in the two fragments is related by an *abstraction relation AR* (to be described below) such that $AR(s,t)$ implies $s \approx t$ for all s and t, and moreover, any corresponding pair of abnormal states is related by a weaker abstraction relation *CR*. This forms a stepwise relationship between the two fragments, as illustrated, for example, in

$$\begin{array}{ccccccccccccc} s_0 & \xrightarrow{\mathsf{r}} & s_1 & \xrightarrow{\mathsf{w}_{a,d}} & s_2 & \xrightarrow{\mathsf{f}} & s_3 & \xrightarrow{\mathsf{w}^{\mathsf{c}}_{a',d'}} & s_4 & \xrightarrow{\mathsf{r}^{\mathsf{c}}} & s_5 & \xrightarrow{\mathsf{r}} & s_6 & \longrightarrow \\ CR & & AR & & AR & & AR & & CR & & CR & & AR & \cdots \\ t_0 & \xrightarrow{\mathsf{r}} & t_1 & \xrightarrow{\mathsf{w}_{a,d}} & t_2 & \xrightarrow{\mathsf{f}} & t_3 & \xrightarrow{\mathsf{w}^{\mathsf{c}}_{a',d'}} & t_4 & \xrightarrow{\mathsf{r}^{\mathsf{c}}} & t_5 & \xrightarrow{\mathsf{r}} & t_6 & \longrightarrow \end{array} \quad (3)$$

Intuitively, the abstraction relation *AR* captures how a normal state in $\mathbb{P}$ is interpreted as a state in $\mathbb{S}$. For example, one part of *AR* describes where the current data at a logical address—i.e., an entry in the *volatile* array of an $\mathbb{S}$-state—can be found in a $\mathbb{P}$-state. On the other hand, for abnormal states in $\mathbb{P}$, only the in-flash data are reliable, and the crash abstraction relation *CR* describes only how the in-flash part of a $\mathbb{P}$-state is interpreted as the stable part of an $\mathbb{S}$-state. Back to diagram (3): The in-memory part of s_0 ($\in \mathcal{N}_{\mathbb{P}}$) is not yet initialized, and thus s_0 only satisfies *CR* with some t_0 ($\in \mathcal{N}_{\mathbb{S}}$). A successful recovery operation brings the disk to a normal $\mathbb{P}$-state that satisfies *AR* with an $\mathbb{S}$-state. This *AR* relationship is preserved by successful regular and flush operations, but deteriorates to *CR* when a regular or flush operation crashes. Recovery attempts may fail but *CR* is preserved, and the relationship is restored to *AR* after the recovery succeeds, from which point we can resume using the disk.

The stepwise relationship is established inductively by showing (i) that initially *CR* holds for all $s \in \mathcal{N}_{\mathbb{P}}$ and $t \in \mathcal{N}_{\mathbb{S}}$ (to establish, for example, the leftmost $CR(s_0,t_0)$ in diagram (3)), and (ii) that each operation preserves *AR* or *CR*, or transforms *AR* into *CR* or vice versa (giving rise to each of the squares in diagram (3)). The inductive cases (ii) are called *type A per-operation correctness*. For example, the type A per-operation correctness formula for the crashed flush operation f^{c} is

$$\forall s,t,s'.\, AR(s,t) \wedge RI(s) \wedge s \xrightarrow{\mathsf{f}^{\mathsf{c}}} s' \implies \exists t'.\, t \xrightarrow{\mathsf{f}^{\mathsf{c}}} t' \wedge CR(s',t') \quad (4)$$

where *RI* is the *representation invariant* describing properties that should be satisfied by the various data structures in a $\mathbb{P}$-state, and is needed in the antecedent to make per-operation correctness provable. For example, a part of *RI* states that the in-memory L2P should agree with the in-flash L2P in addition to all the delta pairs. Like *AR* and *CR*, there is also a weaker version of *RI* called *CI* that describes only the properties about the in-flash part of a $\mathbb{P}$-state, and is used in the relevant per-operation correctness formulae. Note that per-operation correctness is about the behavior of an operation in general, not about its effect on particular states. We have thus reduced reasoning about fragments, of which there is a myriad possibilities, to reasoning about operations, of which there is only a handful.

Finally, to make the induction go through, we need to establish *RI* on all normal states and *CI* on all abnormal states so that the *RI* and *CI* premises in the type A per-operation correctness formulae are satisfied. This is done by showing that the invariants are suitably preserved or transformed by each operation, for example,

$$\forall s,s'.\, RI(s) \wedge s \xrightarrow{\mathsf{f}} s' \implies RI(s') \quad (5)$$

These formulae are called *type B per-operation correctness*.

Up to this point, what we have proven is that $\mathbb{P}$ is behaviorally correct if (i) $AR(s,t)$ implies $s \approx t$ for all s and t, (ii) $CR(s,t)$ for all $s \in \mathcal{N}_{\mathbb{P}}$ and $t \in \mathcal{N}_{\mathbb{S}}$ and $CI(s)$ for all $s \in \mathcal{N}_{\mathbb{P}}$, and (iii) (type A and type B) per-operation correctness holds for each operation. The three conditions are discharged using automatic verification techniques described next in §5.

5 Verifying the SCFTL Implementation

We use the SMT solver Z3 [18] to prove the correctness of the aforementioned three conditions. The first two conditions are easy to check: once we have the formulae describing the invariants *RI* and *CI* and the abstraction relations *AR* and *CR*, we can easily construct the corresponding formulae and let Z3 prove their validity automatically. For the third condition, i.e., the per-operation correctness, we need to construct the formulae $s \xrightarrow{op} s'$ in $\mathbb{P}$ from the C implementation for all operations *op*. We use the symbolic executor Serval [32] to build these formulae (§5.1). If we naively construct the per-operation correctness formulae, often the generated formulae would be too difficult for Z3 to solve. We explain in §5.2–§5.4 how to simplify the formulae so that Z3 can handle them.

5.1 Modeling flash states and crashes

To perform symbolic execution for SCFTL, we need to translate C statements into formulae describing how they update the $\mathbb{P}$-states. A $\mathbb{P}$-state includes a *memory state* and a *flash state*. The memory state is a mapping from variable names to their in-memory value. For example, it maps L2P to an in-memory table. Serval can handle the update of memory states and produce the corresponding formula automatically. We model a flash state as a function $content(bk,pg)$ that maps a *flash location*, which is a pair (bk,pg) where *bk* is a block address and *pg* is a page address, to the data stored in that

page, and modify Serval to support the flash commands SYNC, ERASE, READ, and WRITE. More concretely, we need to tell Serval how those commands update flash states.

The WRITE(bk, pg, d) command updates the flash state $content(b,p)$ to $ite(bk = b \wedge pg = p,\ d,\ content(b,p))$, where $ite(g,\ e_1,\ e_2)$ is a shorthand for "if g then e_1 else e_2." The ERASE(bk) command updates the flash state $content(b,p)$ to $ite(bk = b,\ empty,\ content(b,p))$, where *empty* is a page (which can be modeled as, e.g., an array) with all cells valued -1. Neither the READ(bk,pg) nor the SYNC commands change $content(b,p)$ in the flash state.

Handling asynchronous flash operations The flash commands are *asynchronous*, i.e., the invoked commands first wait in a queue and start to update the flash memory only when the scheduler selects them. Updates to the same page will be executed in the same order in which they come into the queue, but there is no restriction regarding when the updates happen for different pages. If the system crashes, it will lose all commands in the queue.

The flash command SYNC blocks the system until the queue becomes empty. If the system crashes right after a SYNC command, there will be only one possible flash state. However, when it happens between two SYNC commands, there can be multiple possibilities, because we do not know which of those queued commands are processed.

Example 1. Suppose the content at the location (b_1,p_1) is *empty* before invoking the sequence of flash commands WRITE(b_1,p_1,d_1), ERASE(b_1,p_1), WRITE(b_1,p_1,d_2), SYNC. If the system crashes right before SYNC, the content at (b_1,p_1) can be either *empty*, d_1, or d_2.

One way to model crash behavior is to use *crash schedules* [42, §3.1], which are a set of boolean variables representing the occurrence of crash events during the execution of an operation. A special case where all the boolean variables are true indicates a successful execution, in which all the WRITEs invoked by the operation are synchronized. If we adopted this approach, then SCFTL would have to issue a SYNC at the end of each operation, limiting concurrency within a single operation. However, in our SCFTL implementation, it often happens that a SYNC is invoked only after multiple operations. Thus, this modeling would reduce performance significantly.

An alternative approach represents each flash page as a *history of values* [13, §3.2], which are the values written asynchronously to the flash page since the last SYNC. The history can be implemented as a *list*, and a crash non-deterministically chooses a value from the list. This modeling does not require synchronization at the end of an operation, and therefore does not limit concurrency. However, lists are not well supported by SMT solvers.

We propose a novel approach to model crash behavior, which does not limit concurrency and is amenable to SMT reasoning. The main idea is to "over-approximate" possible flash states when they are affected by asynchronous updates. We implement the idea by adding to the flash state a mapping $sync(b,p)$ that maps a flash location (b,p) to a boolean value denoting whether the page is synchronized, i.e., it is not affected by asynchronous updates since the last SYNC. The WRITE(bk,pg,d) command updates $sync(b,p)$ to $ite(bk = b \wedge pg = p,\ false,\ sync(b,p))$ and the ERASE(bk) command updates it to $ite(bk = b,\ false,\ sync(b,p))$. The READ$(b,p)$ command does not change $sync(b,p)$ and always returns $content(b,p)$ no matter $sync(b,p)$ is true or not. The SYNC command remaps all locations of *sync* to *true*.

If *op* is a successfully executed operation, we collect all $\mathbb{P}$-states produced after executing *op* symbolically and use them to construct the formula for $s \xrightarrow{op} s'$.

If *op* is a crashed operation, we collect all possible crash states in two steps. We first collect all flash states (i) right before every SYNC command and (ii) after executing *op*. We then update the $content(b,p)$ function of the collected states to $ite(sync(b,p),\ content(b,p),\ any)$, where *any* means the content can be any value. We model *any* with *fresh variables*, whose values can be arbitrarily assigned. The formula $s \xrightarrow{op} s'$ can then be constructed from all the $content(b,p)$ functions of the collected states. We also designed a suitable crash representation invariant *CI* for SCFTL operations that records flash disk information that is just sufficient for the recovery operation. For instance, it states that "at least one of the two full checkpoints is committed." The *CI* can be guaranteed even with the over-approximated flash state we just introduced.

As said, the formulae produced with the approach we just described may be too difficult for SMT solvers to solve. Below (§5.2–§5.4) we introduce the techniques we use to simplify the formulae and make automatic verification feasible. These techniques are very general and should be usable by other automatic verification projects.

5.2 Crafting the abstraction relations and representation invariants

To avoid overwhelming the SMT solvers, care must be taken to put the abstraction relations and representation invariants in a suitable form. Below we look at a concrete example. A part of the abstraction relations asserts that the stable L2P in $\mathbb{S}$ should agree with its concrete representation in $\mathbb{P}$, which is the in-flash L2P stored in a committed full checkpoint, in addition to all the committed delta pairs stored in the delta region (Figure 2). As opposed to representing the assertion as a relation, we could define a *function* that computes the physical address for a given logical address *la* from a $\mathbb{P}$-state by starting with the in-flash L2P and then sequentially applying the delta pairs, and assert that the stable L2P in $\mathbb{S}$ agrees with the results of the function. Serval compiles the assertion to the following constraint (assuming for simplicity that there are only two delta pairs (la_0,pa_0) and (la_1,pa_1)):

$$\forall la.\ L2P_{stable}[la] = ite(la = la_1,\ pa_1,\ ite(la = la_0,\ pa_0,\ L2P_{flash}[la]))$$

The number of *ite*s is the same as the number of delta pairs, which in the implementation is set to be large enough to avoid frequent full checkpointing. The resultant formula turns out to be too large for the SMT solvers to handle, though.

We thus choose to represent the assertion as a relation, which is divided into two disjoint parts. The first part considers logical addresses that appear in the delta region, and the second part considers those that do not. In more detail:

- For every ($\forall$) logical address la that appears in the region, there exists ($\exists$) a pair (la, pa) in the region such that $L2P_{stable}[la] = pa$. Moreover, this pair should be the last one about la in the sense that for any ($\forall$) subsequent pair (la', pa') we have $la' \neq la$.
- For every ($\forall$) logical address la that does not appear in the region, we have $L2P_{stable}[la] = L2P_{flash}[la]$.

The first part of the relation is still not amenable to efficient SMT solving as it contains *quantifier alternation* of the form $\forall x.\, \exists y.\, \forall z.\, \ldots$, which is often too hard for the SMT solvers [4, 44] to deal with. The key observation is that the existential quantifier $\exists$ can be avoided with the *ghost variable* technique [20]: because pa is determined by la and the $\mathbb{P}$-state, we can extend the $\mathbb{S}$-states with an auxiliary array aux (along with some other ghost variables required to determine pa) and modify the transition relation of $\mathbb{S}$ to keep track of the last pa associated with each la; the formula can then simply use $aux[la]$ (instead of an existentially quantified variable) wherever it needs to refer to the last pa associated with la. Note that the transitions of the non-ghost variables (i.e., *volatile*, *stable*, and *wcnt*) must not depend on the ghost variables (e.g., *aux*), so that the specification of interest (Figure 4) is essentially a *projection* of the $\mathbb{S}$-states, in which ghost variables are removed from the state space.

5.3 Categorizing the invariants

We use the observation that, usually, the invariants and abstraction relations can be grouped into different categories and handled separately in verification. For example, the *RI* may include constraints for different components of SCFTL, e.g., checkpoint and garbage collection. To simplify the presentation, here we assume $RI(s) = RI_{chk}(s) \wedge RI_{gc}(s) \wedge RI_{other}(s)$, where $RI_{chk}(s)$ are invariants related to checkpoints, $RI_{gc}(s)$ are those related to garbage collection, and $RI_{other}(s)$ are other invariants. Now we can divide Formula 5 into three simpler formulae: $\forall s, s'.\ RI(s) \wedge s \xrightarrow{\mathsf{op}} s' \implies RI_x(s')$, where $x \in \{chk, gc, other\}$, and prove their correctness separately. We can further simplify the formulae by using only a subset of the constraints in *RI* to show the preservation of invariants. The reason is that to show $RI_x(s')$ holds after the execution of op, we usually do not need the starting state s to also satisfy the invariants related to other components. Hence we can use the formula $\forall s, s'.\ RI_x(s) \wedge s \xrightarrow{\mathsf{op}} s' \implies RI_x(s')$ instead.

5.4 Partitioning the proofs

Both the implementation and specification of SCFTL involve "non-determinism" when a flush operation crashes. In the implementation, we collect multiple flash states to produce the formula $s \xrightarrow{\mathsf{fc}} s'$. In the specification (Figure 4), when a flash operation crashes, the stable data may remain unchanged ($t'.stable = t.stable$) or change to the volatile data ($t'.stable = t.volatile$). We found that such non-determinism induces verification bottlenecks. We tried to prove the correctness of flush using Z3 with the presence of such non-determinism, but the solver failed to solve it given a few days of time budget. In our experience, this usually means the problem is too difficult for Z3 and needs to be simplified. For SCFTL, such non-determinism can be avoided by *partitioning the proofs*. We need to figure out (i) which flash states of the implementation correspond to the specification $t'.stable = t.stable$ and (ii) which correspond to $t'.stable = t.volatile$. More concretely, assuming that we collect two flash states, represented as $f_1(s)$ and $f_2(s)$, during the execution of the crashed flush operation, we can substitute the transition relations in Formula 4 properly to obtain:

$$\forall s, t, s'.\ AR(s,t) \wedge RI(s) \wedge (s' = f_1(s) \vee s' = f_2(s)) \implies \exists t'.\ \begin{pmatrix} t'.stable = t.stable \\ \vee\ t'.stable = t.volatile \end{pmatrix} \wedge CR(s', t') \quad (6)$$

Instead of directly proving Formula 6, which involves non-determinism ($\vee$), we can use Z3 to prove the following two sufficient conditions separately—if we know that the first flash state corresponds to $t'.stable = t.stable$, and the second corresponds to $t'.stable = t.volatile$:

$$\forall s, t, s'.\ AR(s,t) \wedge RI(s) \wedge s' = f_1(s) \implies \exists t'.\ t'.stable = t.stable \wedge CR(s', t') \quad (7)$$

$$\forall s, t, s'.\ AR(s,t) \wedge RI(s) \wedge s' = f_2(s) \implies \exists t'.\ t'.stable = t.volatile \wedge CR(s', t') \quad (8)$$

Although this technique requires manual inspection of each crash state generated during an operation, it significantly improves the scalability of the verification of SCFTL.

6 Discussion

Using an SMT solver has the advantage that once the formulae are constructed, their proofs are done fully automatically. If some of the constructed formulae cannot be solved in a reasonable time, we apply the techniques mentioned above to simplify them systematically. In total, the representation invariant (*RI*) and the abstraction relation (*AR*) contain 98 conditions; their weaker versions (*CI* and *CR*, respectively) contain 17 conditions. We also use *loop invariants* and an inductive proof rule [23] to handle large loops. With these, all but three conditions can be proven correct.

The three unverified conditions are: (i) after a successful recovery, the L2P table is an one-to-one mapping except for

invalid entries; (ii) after a successful recovery, there is sufficient space to accommodate the follow-up writes and gc_{rl}s; (iii) after a successful flush, there is sufficient space to accommodate the follow-up writes and gc_{rl}s. For the three unverified conditions, we use the *validation* technique [37] to ensure their correctness. The validator itself is formally verified (explained in §6.1) and will notify the user when our SCFTL implementation violates the property. Such notification never occurs in all of our experiments.

The Z3 verification time is 4640 seconds for a 8 GB flash disk and 6302 seconds for a 256 GB flash disk. We choose to verify a particular flash disk size at a time (rather than for all sizes) to reduce the number of quantifiers and thus improve the verification time. We also tried to change other parameters (e.g., several write bounds ranging from 2048 to 20480), the verification time ranges from 1 hour to 2 hours; interestingly, a larger value does not imply a longer time.

Table 1. Lines of code for SCFTL.

Component	Lines of code
SCFTL implementation	950 (C)
Snapshot consistency theorems	652 (Agda [33])
SCFTL specification	22 (Rosette [43])
Invariants & Relations	2010 (Rosette)
Ghost variables	538 (Rosette)
Proof partitions	96 (Rosette)
Flash memory model	232 (Rosette)
Core framework	1048 (Rosette)
Total	3946 (Rosette)

Table 1 shows the lines of code for SCFTL. We count the specification, loop invariants, representation invariants, abstraction relations, ghost variables, and proof partitions as our proof, resulting in a proof-to-implementation ratio of 2.8:1. The total development effort is about 6 person-months; a significant part is devoted to finding (an efficient SMT encoding of) the required invariants and scaling the verification with the techniques we introduced in §5. For the trusted computing base, we assume that (i) the flash memory is free of error, (ii) the verification tools Z3, Agda, and Serval are correct, (iii) the translation from LLVM to machine code is correct, and (iv) the LightNVM [6] Linux kernel module, which we used to host our SCFTL, is correct.

6.1 Validating unverified conditions

For each of the unverified conditions, we implement a validator to monitor if the condition is indeed satisfied during runtime. More specifically, we add to the $\mathbb{P}$-states a set of *validation variables*, including a flag indicating whether the validation fails or succeeds. The validators are not allowed to modify the $\mathbb{P}$-states other than the validation variables. We prove that the validator establishes the following postcondition: if the flag indicates a successful validation, then the condition holds. Although the validation approach is not as useful for storage systems as for compilers, we regard the approach as a last resort to circumvent the limitation of automatic verification.

Validation can also be used to incorporate unverified components into SCFTL. For example, an unverified block allocator may keep track of the block usage and allocate the block with the least amount of usage for wear leveling. A validator can then validate whether the allocated block is actually in the free block queue, and if so, returns the block. Otherwise, SCFTL falls back to the default verified behavior, e.g., allocating the first block in the free block queue.

6.2 Support for concurrency

Our verification methodology does not support concurrent SCFTL operations, and our specification has a sequential nature. However, both of them do not limit an implementation from exploiting the high degree of hardware parallelism commonly seen in modern flash disks (e.g., multiple channels and flash chips). More specifically, executing a single step (e.g., a write operation) in the specification corresponds to performing a top-level C function in the SCFTL implementation. The C function usually uses asynchronous flash commands to avoid waiting for slow flash operations to finish. This design allows multiple flash operations to be executed concurrently until a SYNC. Reordering due to the concurrency of flash operations can only be observed when a crash occurs. We describe our technique to capture reordering in §5.1.

7 Evaluation

To evaluate SCFTL, we conducted experiments designed to answer the following questions:

- Is SCFTL actually correct? (§7.1)
- How does SCFTL perform compared to other FTLs implementing different disk models? (§7.2)
- Is the guarantee of snapshot consistency provided by SCFTL useful to its upper layers? (§7.3)

All experiments were done on a host machine with a 12-core 3.2 GHz Intel i7-8700 CPU and 16 GB of DRAM. To emulate the flash memory, we run experiments on Linux 4.15 hosted by FEMU [29], a QEMU-based emulator that can emulate an Open-Channel SSD (OCSSD). We used liblightnvm [2] with the libaio [1] backend to access the underlying OCSSD in an asynchronous way. The original version of FEMU supports latency emulation for OCSSD commands, but as libaio can only issue NVMe base commands, we followed FEMU's approach to emulate latency for NVMe base commands. We validated that the results produced by the two sets of commands are consistent. The OCSSD has 4 channels, 4 dies per channel, and a total of 8 GB flash memory. We reserved 16 MB of flash memory for the full checkpoint region, 256 MB for the delta region, and set the write bound to 2048.

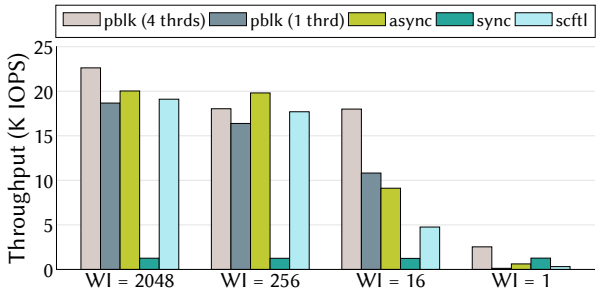


Figure 5. Throughput of FTLs under random writes with different write intervals (WI).

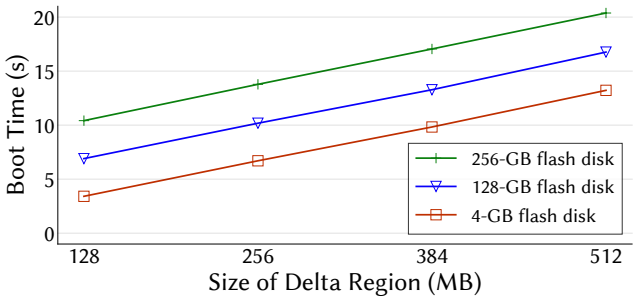


Figure 6. Boot time of SCFTL.

7.1 Stress testing and crash state simulation

To validate the correctness of SCFTL, we designed a testing framework that allows fast stress testing and crash state simulation. The framework hosts SCFTL by emulating the flash memory with DRAM, and simulates a crash by overwriting all pages affected by asynchronous WRITE or ERASE since the last SYNC with garbage data. The framework uses a *workload generator* to issue a sequence of writes and flushes to SCFTL, and simulates a crash based on a given probability. We set a higher probability for configurations that are more likely to result in corner cases (e.g., crashes during recovery).

The framework maintains a pair of arrays, volatile and stable, as golden results, and changes their states according to Figure 4. A *result checker* is then periodically activated to read every sector of SCFTL and check whether the results produced by SCFTL is consistent with the volatile array. To speed up testing, the checker only compares the first few bytes of the read data. We ran the test with 4 configurations for about 8 hours. In total, the workload generator wrote more than 1.4 TB of data, issued about 12 millions of flushes and simulated about 10 thousands of crashes. SCFTL successfully recovered from every crash state and passed all checks.

7.2 Comparing SCFTL with other FTLs

Besides SCFTL, we implemented two additional FTLs with different crash guarantees. The two FTLs implement the asynchronous (denoted by async) and synchronous (denoted by sync) disk models respectively. async is implemented in a way similar to SCFTL, except async (i) does not do checkpointing, (ii) does not comply with 2PGC (i.e., victim blocks are erased immediately after all valid data is relocated), and (iii) has no write count constraint. sync is the same as async, except sync always uses synchronous operations to access the underlying flash memory. We assume the one-page merge buffer of sync is backed by a battery (i.e., data copied to the buffer is guaranteed to be persistent); thus sync can safely ignore any flush request. We also used the state-of-the-art FTL pblk [6], which has similar features to async (e.g., both of them implement the asynchronous disk model and use a sector-level L2P), to understand the quality of our FTL implementation.

We wrote a small program to randomly issue 4 KB writes to a disk, and periodically flush the disk after a fixed number of writes. We call this period the *write interval*. For pblk, we also used a concurrent version (employing 4 threads) of the same program as a way to identify the limitation of SCFTL's sequential nature. Figure 5 shows the average throughput. We first draw two conclusions: (i) Our baseline FTL, async (3^{rd} bars), has a performance characteristic similar to pblk (2^{nd} bars). (ii) Concurrent workloads (1^{st} bars), in general, have higher total throughput than sequential workloads (2^{nd} bars); but the improvement is less obvious when the write interval is higher (e.g., WI = 2048 and WI = 256) because the underlying flash memory has fewer idle resources to serve the concurrent requests. Next, we compare SCFTL with async and sync.

When the write interval is set to 2048, SCFTL throughput is within 5% of async on average; with the write interval set to 256, SCFTL throughput is still within 11%. In both settings, SCFTL outperforms sync by more than 14*x*. With the write interval set to 16, SCFTL throughput drops to 53% of async, but still outperforms sync by 3.8*x*. When the write interval is reduced to 1, SCFTL throughput is only one half of async and one quarter of sync, because SCFTL writes one additional delta page on receiving a flush. Note that the last setting is not a reasonable usage of SCFTL, but it shows the overhead of SCFTL under the worst-case scenario.

We also analyze the write and flush latency and make the following observations: (i) SCFTL has a slightly higher average flush latency (12 ms) than async (10 ms) because SCFTL writes an additional delta page during a flush. (ii) SCFTL has a higher maximum flush latency (398 ms) due to full checkpointing; we can reduce the latency with a hybrid setting (similarly to the WAFL file system [22] which puts a log of requests on non-volatile memory and other data on slower disks), in which the full checkpoints go to memory technologies with a lower latency (e.g., SLC flash and 3D XPoint memory [21]), and other data stays in ones with a higher latency but a lower cost (e.g., MLC and TLC flash).

Finally, Figure 6 shows the boot time of SCFTL. We have not yet implemented optimizations for recovery to reduce the boot time, so the boot time is nearly the same regardless of whether there is a graceful shutdown or where a crash occurs. In general, the boot time is directly proportional to the size of the delta region and the size of the logical address space.

7.3 Modifying xv6 with SCFTL

To understand the usefulness of snapshot consistency guaranteed by SCFTL, we used xv6 [16], a simple *log-based* file system, as our example. In order to prevent any file system inconsistency (e.g., a directory entry pointing to a free inode)

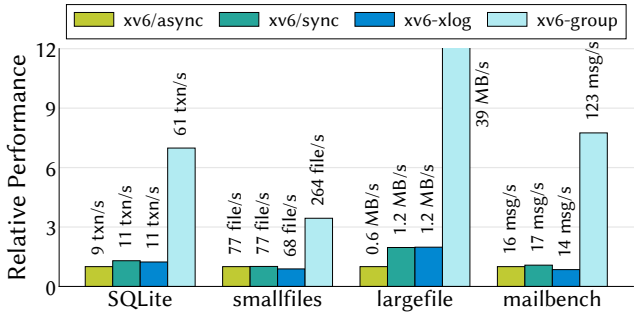


Figure 7. Performance of xv6 on different FTLs. SQLite generates 1K insert transactions followed by 1K update transactions; smallfiles repeatedly creates a file, writes 100 bytes of data to it, and calls `fsync`; largefile writes 4 MB of data to a file and calls `fsync` for every 1 MB; mailbench models a mail server running on the sv6 operating system [14]. To ensure durability, mailbench invokes an `fsync` for each message. We ran each workload 5 times and reported the average. The standard deviation is less than 8%.

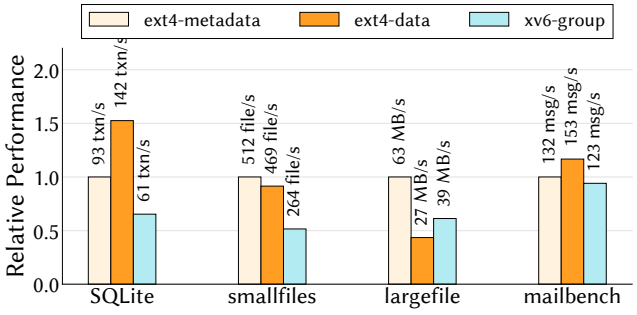


Figure 8. xv6 on SCFTL vs. ext4 on pblk. When running `mailbench` on `ext4-metadata` that does not guarantee the ordering between data and metadata, we invoked one additional `fdatasync` on the temporary file [12, Figure 1].

due to a crash occurring in the middle of a system call, xv6 uses a *write-ahead log* for atomically writing multiple sectors of data to its underlying disk. Such atomicity, however, can be easily achieved with SCFTL. We thus modified xv6 to bypass its log so that data does not need to be written twice, once to the log and once to its actual location. We further modified xv6 to support a common optimization known as *group commit*, which groups multiple system calls into one *transaction*, to reduce the number of flushes. With group commit, xv6 only issues a flush when a transaction is full or on receiving an `fsync`. The implementation is rather easy with SCFTL; we changed less than 30 lines of code of xv6. We compared the two modified versions of xv6 on SCFTL (denoted by xv6-xlog and xv6-group, respectively) with the original xv6 on the asynchronous and synchronous disks used in §7.2 (denoted by xv6/async and xv6/sync, respectively). We used existing file system benchmarks [14, 40] to evaluate the performance.

Figure 7 shows the results. The performance of xv6-xlog is only on par with that of xv6/async and xv6/sync. Although xv6-xlog has reduced the use of writes and flushes via bypassing the log, issuing a flush at the end of each system call would inevitably result in a small write interval, for which SCFTL does not perform very well as shown in Figure 5. xv6-group performs much better than the other three as the write interval becomes larger when multiple system calls are grouped together. The performance difference is particularly obvious for largefile, where `fsync`s are less frequent.

Note that while xv6/async, xv6/sync, and xv6-xlog guarantee *immediate durability*, that is, the result of a system call is successfully stored in the disk after the call returns, xv6-group only guarantees system calls before the last `fsync` are persisted, and system calls after the last `fsync` will not be reordered. This property is also known as *sequential crash consistency* [7]. In practice, sequential crash consistency is a very strong property and is what most application developers actually require [35].

Finally we compare our group commit version of xv6 with the state-of-the-art storage stack: ext4 on pblk. We mounted ext4 with two configurations: The default metadata journaling mode `data=ordered` (denoted by ext4-metadata), and the data journaling mode `data=journal,journal_async_commit` (denoted by ext4-data). ext4-metadata does not journal data but it issues one more flush than ext4-data when committing an ext4 transaction.

Figure 8 shows the results. xv6-group performance is 7% to 49% lower than ext4-metadata. Compared with ext4-data, the performance difference is more divergent. For SQLite, xv6-group performance is only 43% of ext4-data; but for largefile, xv6-group is more than 1.4x of ext4-data. Such divergence can be explained by the behavior of the workloads: SQLite frequently issues `fsync`s and causes the performance of SCFTL to degrade; on the other hand, largefile issues much less `fsync`s and a huge amount of data is written in the journal of ext4-data. We believe the performance difference between our modified xv6 and ext4 is mainly owing to the simplicity of xv6. This can be improved by, e.g., optimizing xv6 with in-memory representations for file system operations [5].

8 Conclusion

We believe that our verified SCFTL brings new opportunities for the design of the storage stack. We demonstrate that starting at a lower-level of abstraction can make verifying crash safety easier while still resulting in an efficient system. Formal specification and verification give the user a clear picture and strong confidence of what he/she can assume while designing the upper layers of the storage stack. Our experimental results show that SCFTL can provide a strong crash guarantee without compromising its performance if upper layers can carefully reduce the flush frequency.

There are several avenues for future work. For instance, we would like to extend the work to cover upper layers, such as file systems or database systems, of the storage stack. With a careful design that fully utilizes the advantage of SCFTL, we believe it is likely that we can obtain a verified and yet efficient upper-layer system. The FTLs used in commercial products usually come with several optimizations, e.g., hot-cold data separation and wear leveling. We plan to extend SCFTL to use those optimizations.

Acknowledgments

We thank our shepherd, Frans Kaashoek, and the anonymous reviewers for their valuable feedback. This work was supported in part by Academia Sinica under grant no. AS-CDA-107-M05 and the Ministry of Science and Technology (MOST) of Taiwan under grant nos. 109-2628-E-001-001-MY3, 109-2222-E-001-002-MY3, 107-2923-E-001-001-MY3, 108-2221-E-001-001-MY3, and 108-2221-E-001-004-MY3.

References

[1] Aio. http://man7.org/linux/man-pages/man7/aio.7.html.

[2] liblightnvm. http://lightnvm.io/liblightnvm/.

[3] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 175–188, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Peter Backeman, Philipp Rummer, and Aleksandar Zeljic. Bit-vector interpolation and quantifier elimination by lazy reduction. In *Formal Methods in Computer Aided Design*, FMCAD '18, pages 1–10, 2018.

[5] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 69–86, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST '17, page 359–373, USA, 2017. USENIX Association.

[7] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 83–98, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 209–224, USA, 2008. USENIX Association.

[9] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '19, page 1054–1068, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 243–258, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Yun-Sheng Chang and Ren-Shuo Liu. OPTR: Order-preserving translation and recovery design for SSDs with a standard block device interface. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1009–1023, USA, 2019. USENIX Association.

[12] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 270–286, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4), January 2015.

[15] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 197–212, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Russ Cox, M. Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2020. https://pdos.csail.mit.edu/6.828/2020/xv6.html.

[17] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 15–28, New York, NY, USA, 1993. Association for Computing Machinery.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08/ETAPS '08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. Inside a verified flash file system: Transactions and garbage collection. In *Revised Selected Papers of the 7th International Conference on Verified Software: Theories, Tools, and Experiments - Volume 9593*, VSTTE '15, page 73–93, Berlin, Heidelberg, 2015. Springer-Verlag.

[20] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, oct 2016.

[21] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3D XPoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[22] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC '94, page 19, USA, 1994. USENIX Association.

[23] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[24] Ben Hutchings. [patch 3.2 027/115] jbd2: fix fs corruption possibility in jbd2_journal_destroy() on umount path. April 2016. https://lkml.org/lkml/2016/4/26/1230.

[25] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 97–108, New York, NY, USA, 2013. Association for Computing Machinery.

[26] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. on Consum. Electron.*, 48(2):366–375, May 2002.

[27] Greg Kroah-Hartman. [patch 4.14 138/267] jbd2: Fix possible overflow in jbd2_log_space_left(). December 2019. https://lkml.org/lkml/2019/12/16/1638.

[28] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST '15, page 273–286, USA, 2015. USENIX Association.

[29] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The case of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, page 83–90, USA, 2018. USENIX Association.

[30] Nancy Lynch and Frits Vaandrager. Forward and backward simulations: I. untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[31] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 33–50, USA, 2018. USENIX Association.

[32] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2009.

[34] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. *ACM Trans. Storage*, 13(3), September 2017.

[36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, page 433–448, USA, 2014. USENIX Association.

[37] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, page 151–166, Berlin, Heidelberg, 1998. Springer-Verlag.

[38] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 147–160, USA, 2008. USENIX Association.

[39] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3), August 2013.

[40] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[41] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: ACID transactions for block storage. *ACM Trans. Storage*, 13(1), February 2017.

[42] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, page 1–16, USA, 2016. USENIX Association.

[43] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 530–541, New York, NY, USA, 2014. Association for Computing Machinery.

[44] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Moura. Efficiently solving quantified bit-vector formulas. *Form. Methods Syst. Des.*, 42(1):3–23, February 2013.

[45] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, page 211–226, USA, 2018. USENIX Association.

[46] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, page 449–464, USA, 2014. USENIX Association.



Storage Systems are Distributed Systems (So Verify Them That Way!)

Travis Hance, Carnegie Mellon University
Andrea Lattuada, ETH Zurich
Chris Hawblitzel, Microsoft Research
Jon Howell, VMware Research
Rob Johnson, VMware Research
Bryan Parno, Carnegie Mellon University

Abstract

To verify distributed systems, prior work introduced a methodology for verifying both the code running on individual machines and the correctness of the overall system when those machines interact via an asynchronous distributed environment. The methodology requires neither domain-specific logic nor tooling. However, distributed systems are only one instance of the more general phenomenon of systems code that interacts with an asynchronous environment. We argue that the software of a storage system can (and should!) be viewed similarly. We evaluate this approach in VeriBetrKV, a key-value store based on a state-of-the-art B$^\varepsilon$tree.

In building VeriBetrKV, we introduce new techniques to scale automated verification to larger code bases, still without introducing domain-specific logic or tooling. In particular, we show a discipline that keeps the automated verification development cycle responsive. We also combine linear types with dynamic frames to relieve the programmer from most heap-reasoning obligations while enabling them to break out of the linear type system when needed. VeriBetrKV exhibits similar query performance to unverified databases. Its insertion performance is 24× faster than unverified BerkeleyDB and 8× slower than RocksDB.

1 Introduction

Software verification promises a fundamentally better way of constructing critical systems: Instead of relying on the spotty coverage provided by run-time testing, verification can mathematically guarantee the functional correctness and even the reliability of software at compile time.

In the context of distributed systems, prior work on IronFleet [29] shows how to combine Hoare logic [23, 31], to reason modularly about the behavior of a single program, with TLA-based [40] state-machine reasoning to show that a collection of nodes running that program behaves according to a high-level functional spec when executing in a failure-prone, asynchronous distributed environment. Both techniques employ general-purpose logic, unlike other work in the area that relies on custom languages or logic [17, 18, 58]. The approach is compatible with a reasonable level of automation, modest-scale implementations (2-3K LoC), and performance within a factor of 2 of unverified code.

We observe that the abstraction of a system as a program interacting with a failure-prone, asynchronous environment applies beyond the domain of distributed systems. Indeed, we argue that a very different domain – storage systems – fits quite naturally into this same abstraction, capturing the asynchrony and nondeterminism of crash safety in such systems. Hence we generalize the IronFleet methodology to this new domain, without introducing a domain-specific logic [12], creating a custom language [2], or changing our system's API and implementation to accomodate verification [54].

We evaluate the success of this generalization by constructing VeriBetrKV, a key-value store based on a B$^\varepsilon$tree [7], a complex but asymptotically-compelling write-optimized data structure. Modern persistent key-value stores use write-optimized data structures, such as LSM-trees [19, 20, 26, 37, 43, 53] and B$^\varepsilon$trees [50], in order to efficiently handle random-insertion workloads, which are common in many key-value-store applications. Write-optimized data structures outperform older key-value data structures, such as B-trees, by orders of magnitude. However, these performance gains come at the cost of a significant increase in code complexity.

TLA-based reasoning lets us prove VeriBetrKV's correct behavior under process crashes and under disk sector corruptions, while Hoare logic allows us to reason independently about the implementation-level optimizations needed for performance.

The resulting system is significantly more complex than the closest prior work, a verified in-memory key-value store [29]. The implementation is over 3× larger, and proving it functionally correct and crash safe requires a multi-level (vs. single-level) refinement proof.

Hence, an important contribution of this work are the techniques we developed to apply automated verification at this new scale. To make verification practical for large-scale software development, we need to balance between exploiting automation and controlling it. Ideally, we want *decisive* automation; i.e., automation that quickly tells us whether our

code and proofs are correct or incorrect. Decisive automation keeps developers engaged and efficient. Unfortunately, since general-purpose verification is undecidable, most automation tools can also "time out". This pessimal outcome takes longer (by definition) and provides less direction to the developer, harming morale and productivity [21, §9.1][29, §6.3.2].

To escape the plague of time outs, we present a concrete development discipline that rapidly squashes time-out prone code. This ensures developers spend the majority of their time in a tight verification development cycle. For example, 98.3% of the definitions in VeriBetrKV verify in less than 10s, enabling development of larger code bases with less effort.

Many verification frameworks reason only about code operating on immutable data structures [11, 49], leaving optimized code generation to a compiler such as Haskell's. Most real systems code relies on explicit in-place updates for good performance to avoid data copies. Some verification frameworks enable reasoning about heap-manipulating code using various methodologies from the PL literature, from separation logic [12, 52] to dynamic frames [29, 30, 32]. These techniques rely on both programmer annotations and relatively heavy-weight automation (e.g., SMT solvers [16]) to determine whether a modification to one portion of the heap may have affected other objects on the heap. Despite our time-out-prevention discipline, we encountered challenges with such automation: while it works well in small instances, as the system grows more complex, the automation slows significantly, reducing developer productivity (in line with prior reports [29, §6.2]).

Drawing on ideas from commercial (Rust [33]) and research [2, 51, 59] languages, we integrate a lightweight linear type system that gives dramatic automation improvements with dynamic frames when additional flexibility is needed. We rewrote some core components of VeriBetrKV from dynamic frames into linearity, reducing our proof burden by 31–37% without impacting performance. We also leverage linear types to emit optimized C++ code.

Ultimately, our evaluation (§7) shows that VeriBetrKV is able to deliver much of the performance gains promised by sophisticated key-value store data structures. On insertions using a hard disk, it outperforms BerkeleyDB by 24×. However, there is still work to be done. Insertions in VeriBetrKV are about 8× slower than RocksDB, a highly-tuned commercial key-value store.

As with any research prototype, VeriBetrKV comes with limitations. One limitation of VeriBetrKV is that it is presently single-threaded; it can exploit I/O pipelining but not CPU concurrency. Second, the guarantees of verification are limited by the Trusted Computing Base (TCB): the top-level spec of a crash-robust key-value store, the spec of VeriBetrKV's interface to the OS and runtime, and the verifier and compiler (Dafny [41] and Z3 [16]). Finally, we focus on safety and functional correctness: we guarantee that the system does not return incorrect results, but liveness (i.e., the guarantee that an operation will complete in finite time) is out of scope.

In summary, this paper makes the following contributions:

1. A method of specifying crash safety in a clean and extensible way that generalizes a verification methodology for distributed systems. As a demonstration of its extensibility, we enhance the specification to include robustness to disk corruption.
2. A discipline for managing automation that supports scalable system development.
3. The integration of a lightweight linear type system within a general-purpose verification language, and a large-scale concrete case study quantifying the impact of the type system as compared to previous approaches.
4. A prototype key-value store demonstrating that our verification methodolgy can scale to handle the complexities of modern key-value-store data structures.

2 Assumptions and Background

We summarize the assumptions underpinning our verification results (§2.1), the verification techniques we employ (§2.2-2.3), and the prior work from which we take inspiration (§2.4).

2.1 Assumptions

Every verified system makes a guarantee that is predicated on a set of assumptions which can be divided into three categories. First, the user must trust the top-level application-facing specification of the contract provided by the system. We provide a succinct (283 lines) specification for VeriBetrKV in terms of a dictionary that, when it crashes, reverts to its state at the most recent client `sync` (§3.1.2).

Second, the system must specify an interface to the environment (i.e., the rest of the world) the codifies assumptions about how that environment behaves. VeriBetrKV's interface is an asynchronous I/O bus that reorders but does not duplicate or, except during a crash, drop messages. VeriBetrKV's environment models a block-level disk and the possibility of arbitrary crashes and torn writes (§3.1)

Finally, we assume the correctness of our verification tools, including the build system that runs our verifier, Dafny [41], on each file. We modify Dafny to emit C++ code (§5.2), so we also rely on the correctness of the C++ toolchain used to produce an executable. These trusted tools are comparable to those in other systems verification efforts; e.g., systems verified in Coq [15] trust Coq, Coq's extraction to, say, OCaml, and the OCaml compiler and runtime. Prior research indicates that each element in such a toolchain can itself be verified [6, 35, 42, 44, 56].

Despite all of these assumptions, several studies indicate that verification is a qualitatively better way of developing software [22, 24, 63], compared with current state-of-the-art code development techniques. These studies found numerous defects in traditional software, but none in verified software, only in the (unverified) trusted components.

2.2 TLA+-Style State-Machine Refinement

State machine refinement is an important tool in verifying asynchronous systems [1, 25, 38]. A high-level, abstract state machine models the desired behavior of a system (say, a key-value dictionary) capturing essential features (inserts update the dictionary; queries probe it) and abstracting away irrelevant details (e.g., efficient indexing and data representation). A concrete state machine adds more details, showing one way to instantiate the abstract machine (e.g., using a hash table to implement the dictionary). A safety proof then shows that the concrete state machine *refines* the abstract state machine, meaning that every execution of the concrete state machine can be mapped to a possible execution of the abstract one. Hence, reviewing the abstract state machine tells the consumer what to expect from the concrete state machine's behavior.

A strength of this approach is that the high-level machine can abstract over asynchrony. If the concrete machine has many concurrently-moving parts, but one can show a refinement (because most transitions in the concrete model do not change its abstract interpretation), then we know the concurrency is irrelevant to the abstract behavior.

2.3 Floyd-Hoare Verification

Floyd-Hoare reasoning [23, 31] is a popular technique for reasoning about the correctness of single-threaded imperative programs. The developer annotates the program's functions with pre-/post-conditions about the program's state when entering/leaving the function, and a verifier checks that these claims hold for all possible inputs and outputs.

Verification tools based on Floyd-Hoare reasoning typically do not consider asynchronous interactions with an environment, which makes it difficult to reason directly about, e.g., program crashes that might occur at arbitrary points during execution. Hence, prior work in storage-system verification introduced a novel version of Floyd-Hoare reasoning, Crash-Hoare logic, to cope with this particular flavor of asynchrony [12].

2.4 Verifying Distributed Systems

In their IronFleet work [29], Hawblitzel et al. show how to compose Hoare logic and TLA+-style state-machine refinement to reason about the safety and liveness of distributed systems. They use Hoare logic to reason locally about a single program's behavior, showing that it conforms to an abstract state machine. They then model the distributed system as another state machine whose state consists of N replicas of the program state machine, along with a network represented as a set of in-flight messages. The system transitions by nondeterministically choosing to allow either one of the N program state machines to advance a step, or the network's state to advance (e.g., by delivering a message). The model captures nodes that can fail-stop and a network that can duplicate, drop, and reorder messages. The top-level verification theorem proves that if the program (e.g., a sharded key-value store) runs in a distributed system as modeled, then its behavior matches a concise app spec – a centralized map.

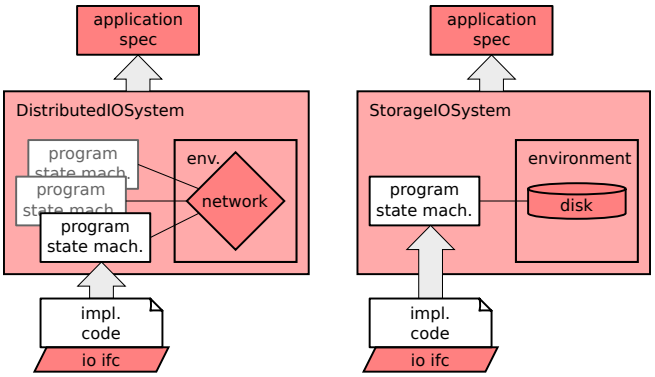


Figure 1: System and environment state machines express how a program interacts with the world to implement an application in IronFleet (left) and VeriBetrKV (right). Shaded shapes are trusted; unshaded shapes are untrusted code and proof.

3 Verifying Storage Systems

To verify storage systems, we observe that embedded within prior work on verifying distributed systems (§2.4) is a general-purpose framework for verifying code that interacts with an asynchronous environment, such as a storage system. This is exciting because it suggests we can use a common methodology to solve a broader class of systems verification problems, and it obviates the need to develop a specialized logic or proof framework for each environment. In this paper, we showcase an instantiation of this general methodology for an asynchronous environment with a single disk and a single processing node. However, we conjecture that the approach generalizes to a variety of systems including multi-node, multi-disk storage systems; indeed, our predecessor, IronFleet [29], has already shown how to model multiple nodes connected by an asyncronous network. Further systems applications might include heterogeneous hardware or device drivers.

In this general framework (Figure 1), the developer uses Hoare logic to prove that their optimized imperative program code, when run synchronously and without crashing, complies with an abstract *program state machine* (e.g., in prior work, this was the code running on each node in the distributed system). The program code uses a trusted API (e.g., for the network) to interact with the outside world, and an important aspect of the refinement proof is showing that the code's interactions with this API match those specified by the program state machine. A second state machine, the *environment*, encodes assumptions about the rest of the world – the parts the program does not control (e.g., the network). A final state machine, the *IOSystem*, composes these two state machines (the program and the environment) and dictates how they interact (e.g., prior work composes N programs running asynchronously, communicating only via the environment's network). The developers prove a top-level theorem showing

that the IOSystem state machine refines a simple, high-level application spec (e.g., the distributed key-value store behaves like a centralized map). This theorem guarantees correct execution in practice if the environment behaves as modeled, if the hosts run an executable matching the program, *and* if the combined elements interact as assumed by the system.

In this work, we instantiate this general framework to reason about storage systems by modeling an environment that contains a disk and in which crashes may occur.

3.1 An Environment Model with Crashes

Our model of a storage system is a special case of the IOSystem as described above. It contains two parts: **(i)** the program, which models the executable VeriBetrKV, and **(ii)** the environment: a model of the disk with an I/O bus. Together, these two components form a *StorageIOSystem* (Figure 1).

The state machine representing this StorageIOSystem can transition in three ways. First, the program state machine can transition forward a step, possibly interacting with the I/O bus. Second, the environment can transition, e.g., by having the disk process a read- or write-command from the I/O bus.

Third and finally (and unlike IronFleet DistributedIOSystems), the StorageIOSystem as a whole can perform a *crash*, which models, e.g., a power failure or a kernel panic. Crashing results in a disk state that remembers every write it has acknowledged, but the data at the address of any unacknowledged write may reflect the old value, the newly written value, or a corrupt value (§3.1.1). For the program, a crash simply resets its state machine to its initial (boot-up) condition.

Our top-level proof shows that the StorageIOSystem is a refinement of the application spec (§3.1.2), demonstrating that VeriBetrKV's top-level guarantees are maintained despite an arbitrary number of crashes occurring at nondeterministic times (outside the program's control).

3.1.1 Corruption

Our disk model allows the disk to corrupt its data at any time (i.e., not just during a system crash). There is only one constraint: corruption cannot produce a block with a valid checksum.[1] When VeriBetrKV detects an invalid checksum, it aborts the current query. This ensures correctness, since VeriBetrKV will not return an incorrect query result. This assumption about the disk's behavior is what checksummed storage systems already (informally) assume. We formalize this assumption and make use of it in our correctness proof.

Using checksums means that VeriBetrKV protects against "torn writes", where a block of the disk is changed to have neither its old value nor the value written, as well as random media corruptions. Of course, some disks do violate our checksum assumption. An adversarially-controlled disk could easily return corrupted data yet with valid checksums. Likewise, a disk which returns stale blocks would also not satisfy our checksum assumption. In either case, it would be impossible for *any* implementation to meet our application spec as written.

However, our methodology provides flexibility in specifying the precise assumptions made about the disk. In principle, an engineer could provide a weaker disk model, and in addition, either provide a cleverer implementation or a weaker application spec to match.

[1] Therefore, our checksum routine, CRC32C [8], exists within our TCB, so that it can be referenced by our disk model.

3.1.2 Application Specification

To achieve good performance, a practical storage system cannot afford synchronous writes. Instead, the application calls `sync` when it requires durability; data not synced is permitted to be lost during a crash. The nondeterministic relationship between nondeterministic `crash` and the application `sync` API must appear in the theorem; each is a transition in the application spec state machine.

Intuitively, the application spec of VeriBetrKV says that in the absence of crashes or block corruption, VeriBetrKV acts exactly like a dictionary, always returning the most-recently-written value. In the presence of a crash, VeriBetrKV is allowed to forget data, but no farther back than the most recent sync. Furthermore, "forgetting" is equivalent to the entire dictionary reverting to a consistent, previous snapshot; future crash-free operations proceed forward from this snapshot. Contrast this promise with a filesystem with crash-corrupted metadata: the data may appear complete and valid, but future operations may result in behaviors that violate the filesystem's contract.

Our storage system specification is easy to state, clean, and easy to utilize from a client application. In contrast, contemporary file systems, for performance reasons, decouple `sync` operations on metadata from `sync` on file data. Such guarantees are very difficult to utilize from a client, and in fact difficult to even state precisely; much of DFSCQ is dedicated to stating such a guarantee precisely [11].

3.2 Refinement Verification Techniques

Our methodology requires a proof of a TLA-style state-machine refinement (§2.2) between a StorageIOSystem and our application spec. Due to the complexity of this proof, we break the refinement into a sequence of smaller refinements (§6.2.1). We use the following techniques to organize the proof, separating concerns between the caching subsystem, the journal subsystem, and B$^\varepsilon$tree manipulations.

State-machine composition. In many cases, we define a templated state machine $S\langle T\rangle$ in terms of an abstract subcomponent T. A refinement T' of the subcomponent T can be lifted to a refinement $S\langle T'\rangle$ of the larger state machine $S\langle T\rangle$. This allows us to build up our refinement in terms of refinement proofs for the subcomponents. For example, a B$^\varepsilon$tree tree refines a simple dictionary spec; therefore, a *crash-safe* B$^\varepsilon$tree refines a *crash-safe* dictionary spec.

IOSystem Refinement. Our proof re-uses the concept of an

IOSystem at several layers of abstraction. For example, at the lowest layer, we model the disk as storing sequences of bytes, but at higher layers, we model the disk in a "type-safe" way, as a collection of nodes. But at all layers, the overall system follows the rules of an IOSystem state machine.

Transposition. High in the abstraction stack, the disk is used in different ways by different modules. For example, the journal models the disk as an array of journal entries, whereas the B$^\epsilon$tree models the disk as storing nodes. At a low level, all the modules together interact with a single, byte-oriented disk. Transposition arguments enable us to split up one StorageIOSystem into multiple, so each can be reasoned about independently.

4 Disciplined Automation

A productive verification workflow uses the developer's time efficiently. This has two aspects: how much tedious typing the developer has to do, and how quickly the verifier replies to the developer's proof attempts.

Functional verification of arbitrary software is undecidable, and hence requires either a limit to expressivity [49, 54, 64] or some degree of manual guidance. In interactive theorem provers [15, 47], developers manually use tactics to tell the prover which steps to take next.

A large verified system has a large number of definitions, such as invariants and state-machine transition relations. An automated program verifier is a great fit for systems verification because so much of the verification work is tedium amenable to automation. Exposing all the definitions to the theorem prover, however, gives the theorem prover a large search space, which can take a long time to explore.

The development cycle is a balance between exploiting automation and controlling it. Faced with a verification failure, the developer must first supply any guidance not provided by automation. That process terminates when the proof passes (because the failure was a weakness of the automation) or the developer discovers an actual flaw. If automation is too weak, the developer burns time tediously typing in the missing proof guidance. If the automation heuristics are too aggressive, the developer burns time waiting for replies from the verifier.

As the system grows, the risk of timeouts grows. We have found it essential to resolve timeouts as soon as they crop up, before there are so many they are difficult to sort out. If a developer observes a $> 20s$ response time, they are expected to stop work and instead resolve the timeout,

The key technique to remedy timeouts is to control how much information the prover has when trying to verify a method or lemma, typically by making fewer definitions visible to the theorem prover. Developers can mark Dafny definitions `opaque`, for example, so that the definitions are hidden by default, except where the developer chooses to explicitly `reveal` the definitions. We use a command-line SMT profiler to pinpoint problematic definitions, i.e., those the solver instantiates too many times in its attempts to construct a proof. Table 7.1 shows that we have followed this discipline with some consistency, and timeouts in VeriBetrKV remain rare.

5 Language Improvements

Verifying low-level systems software means verifying stateful code with in-place updates. Unfortunately, reasoning about updates is painful in the presence of aliasing. Traditional verification tools like Dafny and VCC [13] rely on an SMT solver to reason about aliasing and ownership. For example, Dafny uses dynamic frames [32], where programmers annotate methods with `modifies` clauses to specify which objects each method may modify. With dynamic frames, programmers can write arbitrarily complex expressions to compute the set of modified objects. Programmers can also write arbitrarily complex preconditions and postconditions to specify non-aliasing, usually by specifying the disjointness of various sets of objects used in various modifies clauses. The SMT solver must then reason about these arbitrarily complex expressions, which provides programmers with great flexibility, but painfully slows verification [29, §6.2]. Furthermore, this reasoning is mixed with reasoning about functional correctness properties, often making it confusing for the developer to diagnose errors: does a verification failure indicate something deep about the invariants and states, or just a missing non-aliasing requirement?

Recent work on low-level type-safe languages like Rust [33] point to an alternate strategy, where the language's type checker quickly takes care of memory safety and ensures non-aliasing. Full tools for verifying Rust-like programs [3] are still under development and are not yet as mature as tools like Dafny, Coq, and VCC. Therefore, we use Dafny as a starting point and extend it in a more Rust-like direction in two ways. First, rather than using Dafny's existing high-level code-generation backends, we wrote a C++ backend for Dafny that generates efficient C++ code that does not require a garbage collector (§5.2). Second, we extended Dafny's static type checker to support linear variables (§5.1), which allow us to reason purely functionally about data that can be mutated and manually deallocated. This extended type checking needs no SMT solving and therefore places no burden on the SMT solver. Section 6.1 describes the use of linearity in our implementation. Section 7 quantifies the dramatic reduction in proof code it produces and confirms that our use of linear reasoning has a negligible impact on run-time performance.

Our approach to linearity combines ideas from linear type systems [59] like Cogent [2], linear variables in CIVL [36, 51], and Rust's ownership borrowing. Crucially, our extended type system integrates with Dafny's existing dynamic frames, so that we can use linearity to speed verification where possible and fall back to dynamic frames when we need more flexibility. This allows us to verify the safety of highly-aliased code that would require run-time checks or unsafe code in Rust, or would fall outside Cogent's linearity restrictions.

```
function method seq_get<A>(shared s:seq<A>, i:uint64) : (a:A)
function method seq_set<A>(linear s1:seq<A>, i:uint64, a:A)
  : (linear s2:seq<A>)
function method seq_free<A>(linear s:seq<A>)

method M(a:array<uint64>, o:seq<uint64>,
         linear l:seq<uint64>, shared s:seq<uint64>) ... {
  linear var l2:seq<uint64> := l; // ok: consumes l
  linear var l3:seq<uint64> := l; // error: l already consumed
  var n:uint64 := seq_get(l2, 10); // ok: borrows l2
  l2 := seq_set(l2, 10, 20); // ok: consumes l2, then restores l2
  seq_free(l2); // ok: consumes l2
  seq_free(l2); // error: l2 already consumed
}

class BoxedLinear<A> {
  function Has():bool
  method Give(linear a:A)
    modifies this; requires !Has(); ensures Has(); ...
  method Take() returns(linear a:A)
    modifies this; requires Has(); ensures !Has(); ...
  function method{:caller_must_be_pure} Borrow() : (shared a:A)
    requires Has(); ...
}
```

Figure 2: Using linearity in extended Dafny

5.1 Linear Variables

Since aliasing and mutation are expensive to reason about, we use linearity to express non-aliasing or non-mutation. Specifically, we extend Dafny with a keyword `linear` to express non-aliased, mutable values, and a keyword `shared` to express aliased, immutable values.

Figure 2 shows example code written in our extended version of Dafny. Dafny can express both purely functional code, with no heap modification, and imperative code that allocates and modifies heap data. A Dafny `method` can perform both functional and imperative operations, while a `function method` can perform only purely functional operations. The method `M` demonstrates various kinds of Dafny variables. The variables `a` and `o` use existing Dafny features: `a` is an ordinary pointer to a mutable array in the heap, and `o` is an ordinary immutable sequence. `a` and `o` rely on garbage collection (in C#) or reference counting (in C++) for memory management. The variables `l` and `s` use our extensions to Dafny: `l` is a linear (non-aliased) mutable sequence, and `s` is a shared (potentially aliased) immutable sequence temporarily borrowed from a linear sequence. `l` and `s` do not rely on garbage collection or reference counting. (The declarations `linear l:seq<uint64>` and `shared s:seq<uint64>` are similar in spirit to Rust's declarations `l:&mut[u64]` and `s:&[u64]`, although in Rust's semantics, `l` and `s` are references to sequences, while in Dafny's semantics, `l` and `s` are sequence values, not references.)

The static type checker flags any attempt to duplicate or discard a linear variable like `l` as a type error. In Figure 2, for example, the attempts to assign `l` to both `l2` and `l3` is a type error, as is the attempt to free `l2` twice. The lack of duplication allows efficient in-place updates at run-time, as shown in the call to `seq_set`, which sets one element of a sequence. Despite its efficient implementation, though, the verifier can reason about the call to `seq_set` in a purely functional way [2, 59], without worrying about aliasing and the state of the heap. Like Rust and Cogent, our extension to Dafny allows temporary immutable borrowing from a linear variable, as shown in the call to `seq_get`. Borrowed values are tagged as `shared`, and `shared` variables cannot be returned out of the scope of borrowing.

We also extended Dafny to support linear fields in data structures and linear elements in sequences. In contrast to purely linear systems [2], our system can verify the interoperation between the linear data structures and ordinary heap data (like `array`). First, it supports linear-to-ordinary references: linear data structures can hold ordinary fields and sequence elements, such as Dafny heap pointers and arrays.

Second, to support ordinary-to-linear references, our extension provides a novel trusted class `BoxedLinear<A>`, shown in Figure 2, which stores linear values in ordinary objects. Each `BoxedLinear` object is an ordinary heap object, and references to the object can be freely duplicated, allowing complex aliasing. However, to take the linear value out of a `BoxedLinear` object, the program must prove that the object currently has the linear value. Taking the linear value sets `Has` to false, so that a program can't take the same linear value more than once. This prevents the linear value from being duplicated. In addition, pure functional code (`function methods`) can `Borrow` directly from `BoxedLinear` without modifying `Has`. Restricting the scope of the borrowed value to functional code ensures that no imperative `method` can make `Has` false during the borrow. This approach shows the power of combining SMT solving with type system linearity: linear variables bring economy and clarity to the common cases, while SMT reasoning allows greater flexibility (e.g. using lemmas to prove `Has() == true`) when needed.

5.2 Compiling to C++

Dafny compiles its code to C#, Java, JavaScript, and Go. When building a storage system, however, we want more control over memory layout and management than what these high-level, garbage-collected languages offer.

Hence, we have added a new C++ backend to Dafny. We compile Dafny's immutable datatypes to C++ `structs`, and Dafny's classes and arrays to reference-counted pointers to their C++ equivalents. We implement Dafny's immutable sequences using a C++ `struct` that contains a shared pointer to the underlying values, along with an offset and length into those values. This allows us to optimize operations that extract portions of a sequence; because sequences are immutable, it is

safe to implement such operations by returning a new `struct` pointing at the same underlying values but with a different offset and length, rather than copying the values.

When compiling linear variables, we perform updates in-place, rather than copying. Since linear variables cannot be silently discarded, we can rely on explicit deallocation (e.g., `seq_free`) and do not need to reference count them.

Finally, the backend deliberately does not compile Dafny's mathematical integers; it expects the programmer to use Dafny's refinement types to define machine integers that can be (provably) safely compiled to standard C++ integer types.

6 VeriBetrKV: A High-Performance, Verified Key-Value Store

We present a high-level overview of VeriBetrKV's implementation (§6.1) and the structure of its safety proof (§6.2).

6.1 VeriBetrKV's Implementation

VeriBetrKV implements a copy-on-write B$^\epsilon$tree with a logical journal for efficient syncs.

6.1.1 B$^\epsilon$Tree Background

A B$^\epsilon$tree [7] is a write-optimized structure that combines ideas from B-trees and LSM-trees to dramatically improve insertion performance versus a B-tree.

For our purposes, B$^\epsilon$trees have two key properties:

- They support random insertions an order of magnitude faster than B-trees. They achieve this speedup by accumulating newly inserted key-value pairs high in tree and "flushing" items from parent to child in large batches.
- They typically use much larger nodes than a B-tree. B$^\epsilon$tree nodes are often in the range of 1-4MiB, whereas B-tree nodes are in the range of 4-64KiB. This is because B$^\epsilon$trees perform node updates in batch, and hence can afford to update large nodes without incurring high write amplification. As a consequence, queries in an "off-the-shelf" B$^\epsilon$tree are slower than in a B-tree, since each cache miss must read a larger node. Production B$^\epsilon$trees contain optimizations to overcome this problem.

See Bender, et al. [4, 5] for a full exposition of B$^\epsilon$trees and an analytical framework for analyzing their performance.

In VeriBetrKV, we use 2MiB nodes on hard disk, 128KiB nodes on flash, and a fanout of 8.

6.1.2 Node-Buffer Data Structures

Each node in a B$^\epsilon$tree contains a buffer of key-value pairs. VeriBetrKV has two representations for in-memory nodes: a serialized format and an in-memory search-tree format. The former avoids marshaling and demarshaling costs for nodes low in the tree, which are updated through batch flushes. We use the search-tree representation only for the root node, which must support single insertions of new key-value pairs from the user.

The in-memory search tree is one of the VeriBetrKV's most performance-critical components. Thus, we originally wrote it using Dafny's dynamic-frames heap reasoning, making it one of the most difficult pieces of code in our implementation.

We then rewrote it using our linear type system (§5). From the verifier's perspective, this eliminated all heap-mutating code. Furthermore, the type system gave immediate feedback on linear typing errors, enabling rapid development. Section 7 quantifies the reduction in proof code and shows that the shift to linear reasoning had no noticeable impact on performance.

The in-memory search tree also demonstrates the utility of the integration between our linear type system and Dafny's builtin Floyd-Hoare reasoning. Each node in our in-memory search tree maintains a linear sequence of linear (pointers to) children. When we split a node, we need to copy half of those child pointers to the new left-hand node, and half of them to the new right-hand node. In a standard linear type system, such as in Rust, we cannot "take" a subset of the values out of a linear array, and we would have to resort to unsafe code or incur the run-time overhead of using an `Optional` type for each array element.

With our linear type system, each linear sequence `s` has an associated boolean ghost[2] sequence, `lseq_has(s)`, that serves the same purpose as the `Has` predicate of the `BoxedLinear` class in Figure 2. When we remove the child pointers for the new left-hand node the ghost sequence becomes `false` for each index `i` that we take. However, this does not prevent taking the remaining children for the right-hand node, since Dafny infers that `lseq_has(children)[i]` is still `true` for those indices.

6.1.3 Caching, Copy-on-Write, and Indirection Tables

VeriBetrKV maintains a cache (BlockCache) of recently accessed nodes, using an LRU eviction policy. The cache is free to write back a node at any time, which is safe because VeriBetrKV uses copy-on-write. The BlockCache is oblivious to the data structure it caches. It resembles a kernel buffer-cache, except **(a)** its allocation unit is a type parameter so we can allocate tree nodes, **(b)** it tracks inter-block references and garbage collects blocks.

VeriBetrKV implements crash safety by maintaining three copy-on-write B$^\epsilon$trees: a *persistent* tree, a *frozen tree*, and an *ephemeral* tree. New inserts go into the ephemeral tree, the frozen tree is in the process of being made durable, and the persistent tree is the previous tree that was made durable.

Each tree is defined by an *indirection table*, which maps logical node IDs to physical disk addresses. Parents refer to children by logical node ID, and the cache is indexed by logical ID. Since nodes are large, the indirection table is small. For example, the indirection table for a 1TiB disk is only 24MiB.

To sync the tree to disk, we write out all dirty nodes, write the indirection table, and then write a superblock pointing to the indirection table. VeriBetrKV keeps two superblocks,

[2] i.e., a data structure used for proof purposes only, not compiled code

alternating between them, and using a counter to detect which one is newer. We call this process a *checkpoint*.

VeriBetrKV ensures that checkpoints capture a point-in-time-snapshot as follows. When we begin a checkpoint, all dirty nodes in the live indirection table are marked as "write-back-before-editing". Since these nodes are now also referenced by the indirection table of the in-progress checkpoint, we must write them to disk before modifying them in cache. Note that we need not wait for the write to complete: if the system crashes before the write completes, then the system will boot from the previous checkpoint.

The indirection table in VeriBetrKV is implemented as a linear-probing hash table. As with the in-memory search tree, we initially implemented this using Dafny's dynamic frames. In order to isolate the complexities of heap reasoning, we essentially wrote the hash table twice. The first version used immutable data structures, and served as a precise, low-level description of the hash table's behavior. We then wrote it a second time using mutable arrays, proving that each step exactly followed the algorithm in the functional model. This approach separated the proof of functional correctness from the proof of correct heap manipulation, but it meant implementing the hash table twice.

We then reimplemented this hash table using our linear Dafny extensions. In this version, the low-level functional model of the algorithm, suitably annotated with `linear` and `shared` keywords, *is* the implementation, cutting the amount of code substantially (§7).

6.1.4 Optimizing Syncs with a Journal

When applications perform frequent syncs, writing out every dirty node for each sync becomes expensive. For example, if an app performs a sync after every insert, then each sync must write out the root node (2MiB) to persist a single insertion.

We solve this problem with a journal of logical operations. Each insertion is recorded in an in-memory journal as well as inserted into the B$^\epsilon$tree. When the application requests a sync, we simply write the in-memory journal to disk.

Journal space is reclaimed as part of a checkpoint.

6.2 VeriBetrKV's Proof

VeriBetrKV weaves several modularity techniques together to manage the complexity of the code and its correctness argument. We use modular Hoare logic to reason about implementation code and reusable templated state machine models and IOSystem composition (§3.2) to reason about how that code behaves in an asynchronous environment. IOSystems generalize well to this new context of storage systems, and reuse well as we develop a correctness argument up through layers of abstraction. Overall, this blend of techniques is sufficient to modularize the complexity of a modern high-performance storage architecture.

6.2.1 VeriBetrKV's Refinement Proof

Below, we describe how simple state-machine refinement suffices to reason about the correctness of our B$^\epsilon$tree assuming it existed in memory on a crash-free machine (§6.2.2).

To manage complexity, we modularize our proof by separating the reasoning about the journal subsystem and the crash-safe B$^\epsilon$tree storage subsystem. We further modularize the B$^\epsilon$tree proof by separating caching logic from B$^\epsilon$tree manipulation logic.

6.2.2 Simple State-Machine Refinement

State-machine refinement enables designers to organize high-level correctness arguments in isolation from its low-level details, and then ignore abstract concerns when writing implementation code [40]. For example, suppose we want to prove that a single, unfailing process correctly implements an in-memory B$^\epsilon$tree (① in Figure 3).

- The application spec is a *Map* that updates and queries a key-value relation.
- An *abstract* B^ϵ*tree* inserts messages into nodes arranged in a tree, where each node is an infinite map. This model has enough detail to define query semantics over that tree, but the (unimplementable) nodes elide detail that is addressed below.
- The B^ϵ*tree* defines the node data structure that clumps the infinite key range into finite buckets at pivot keys.
- *Impl*B^ϵ*tree* is compilable real imperative code, organized with Hoare logic, with details like mutable data structures and 64-bit integer overflow (gray in the diagram because VeriBetrKV does not have a purely in-memory B$^\epsilon$tree).

The refinement arrow between the B^ϵ*tree* and the *abstract* B^ϵ*tree* concerns only the relationship between pivot nodes and infinite-map nodes; it ignores higher-level concerns (the tree shape) and lower-level concerns (efficient data structures). This application of refinement gives the developer the freedom to modularize the correctness argument.

6.2.3 VeriBetrKV's IOSystem Refinement

Of course, VeriBetrKV's B$^\epsilon$tree does not necessarily run without crashing, and it interacts with an asynchronous byte-level disk, so that the full data structure is stored on disk but cached in memory. Hence, to prove its safety, we repeatedly apply the techniques from §3.2 to prove that the storage IOSystem state machine (§3), when instantiated with VeriBetrKV's program state machine, refines the application-facing specification (§3.1.2).

A good specification of a crash-safe system needs to be able to describe how data is recovered upon crash. Our specification has an *ephemeral* state and a *persistent* state. All user operations (e.g., queries and inserts) are applied to the ephemeral state; if there are no crashes, the user will observe the behavior of a simple dictionary. On a crash, the ephemeral state reverts to the persistent state; the persistent state therefore represents the state made persistent to disk. Background operations can

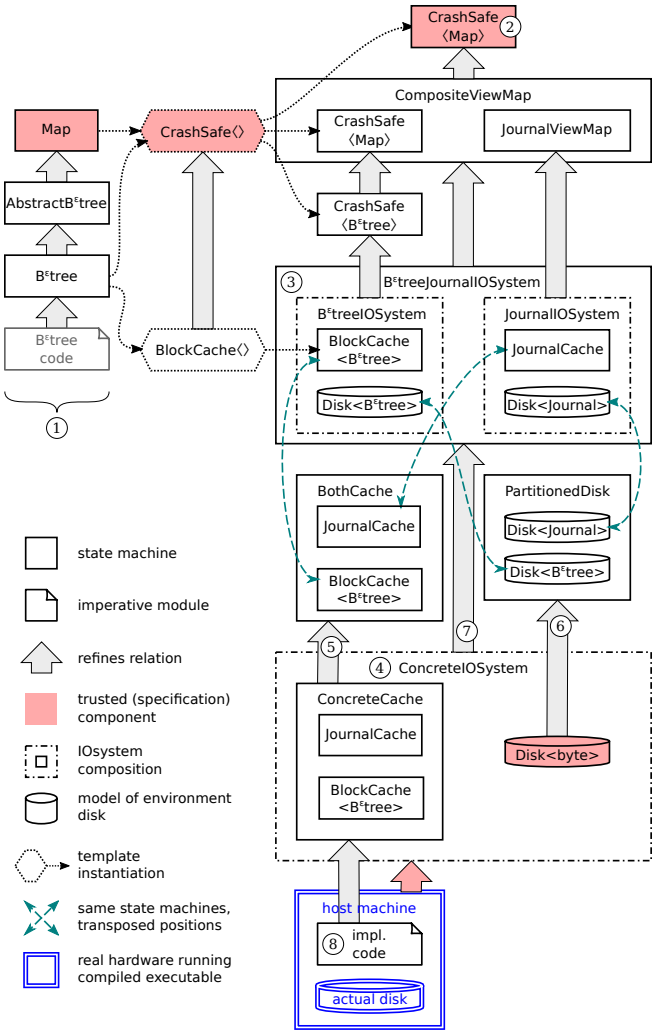


Figure 3: Structure of VeriBetrKV's correctness argument.

update the persistent state to a newer state. Two transitions, `sync_start` and `sync_end` are defined such that, when `sync_end` runs, the persistent state will have updated to the ephemeral state from the `sync_start` (or a newer version). We call this generic specification CrashSafe⟨T⟩, parameterized over a state machine T. In our case, the top-level specification will be CrashSafe⟨Map⟩.

The CrashSafe⟨T⟩ template is an abstraction of our BlockCache⟨T⟩ template state machine, which interacts with a disk but is oblivious to the data structure it caches. We prove generically that if concrete type T_{conc} refines an abstract type T_{abs}, then an IOSystem containing a BlockCache⟨T_{conc}⟩ and a disk refines a CrashSafe⟨T_{abs}⟩.

We apply this generic result at the next level down ③, where we instantiate the type-oblivious BlockCache⟨T⟩ template with the IO-oblivous Bᵋtree (§6.2.2). Leveraging refinement ①, this proves that BᵋtreeIOSystem refines CrashSafe⟨Bᵋtree⟩ and hence CrashSafe⟨Map⟩.

In a sibling library, we prove that the JournalIOSystem refines JournalViewMap, an abstract summary of journal behavior, including components for journal entries persisted to disk, journal entries being written, and journal entries in memory. Ordinary state machine composition pulls those together into the CompositeViewMap, an abstract summary of the state-machine composition BᵋtreeJournalIOSystem. The CompositeViewMap is shown to implement a CrashSafe⟨Map⟩: an abstraction function applies updates in JournalViewMap's journals to the map states in CrashSafe⟨Map⟩ to obtain the application-spec CrashSafe⟨Map⟩.

Thus, we have refined from the application spec to a model ③ of the VeriBetrKV's two main components, each modeled as separate systems, each of which uses a high-level "disk" that stores its client's internal datatype representation.

One layer down, the ConcreteIOSystem ④ introduces a real byte-level disk-IO interface. The program component of this IOSystem is the ConcreteCache, which includes marshaling and demarshaling functions on top of the JournalCache and BlockCache⟨Bᵋtree⟩. The ConcreteCache refines ⑤ the BothCache. The disk component of the ConcreteIOSystem is our low-level disk model, the Disk⟨byte⟩, which refines ⑥ PartitionedDisk via the same marshaling functions. This refinement relies on an invariant that Disk⟨Journal⟩ and Disk⟨Bᵋtree⟩ access mutually-disjoint regions of the disk. With these two refinements in place, we transpose the four subcomponents (dashed green arrows) and obtain a refinement ⑦ from ConcreteIOSystem to BᵋtreeJournalIOSystem. This rearrangement is crucial to allow us to reason about BetreeIOSystem and JournalIOSystem separately.

ConcreteCache is the lowest-level model of the program, including all of the components (Bᵋtree, journal, indirection table, byte-level IO interface). It remains to show that our imperative heap-mutating code ⑧ refines ConcreteCache. To do so, we show that each handler invocation in the code corresponds to a `Next` transition in the ConcreteCache state machine. Because ConcreteCache is a low-level model, this task decomposes nicely along Hoare-logic call-graph boundaries: calls to update the journal advance the JournalCache sub-component, leaving the Bᵋtree unchanged, and vice versa.

6.2.4 VeriBetrKV's Floyd-Hoare Proof

The top-level API methods of VeriBetrKV's implementation (⑧) use Floyd-Hoare logic to show that their operations correspond to transitions of the ConcreteCache state machine. Of course, the ConcreteCache is only one component of the ConcreteIOSystem (④), and likewise, the implementation code interacts with the disk only via a trusted interface.

At the implementation level, we do not reason about the disk itself—we reason only about interactions via the trusted interface. Transitions of the ConcreteCache state machine are labeled with *disk ops*. Each disk op is either a *no-op* (i.e., no disk interaction), an I/O *request*, or an I/O *response*. The disk-op labels are "visible" to the ConcreteIOSystem: the ConcreteIOSystem state machine is defined in terms of the

Major component	spec	impl	proof
Map, CrashSafe⟨Map⟩	283	82	818
AbstractB$^\varepsilon$tree	0	70	2024
B$^\varepsilon$tree	0	137	7079
CompositeViewMap	0	26	823
B$^\varepsilon$treeIOSystem	0	246	6510
ConcreteIOSystem	270	68	2887
implementation code	180	5380	21697
libraries	477	364	2847
total	1210	6373	44685

Table 1: Line counts [60] by major components in Figure 3.

	hash table		search tree	
Aliasing reasoning	impl	proof	impl	proof
Dynamic frames	289	1678	289	2220
Linear type system	289	1063	373	1531

Table 2: Line counts [60] of two subcomponents, comparing dynamic-frame implementations with our linear type system.[3] Linear typing reduces the proof burden by 31–37%

ConcreteCache's interaction with the disk via disk ops.

Thus, the Floyd-Hoare logic in our implementation does not need to reason about the disk proper. Rather, it only needs to show that each interaction with the trusted disk interface corresponds to a disk op which is valid according to the ConcreteCache state machine.

Overall, our proof shows that an executable built from our implementation's code, if run on a real host with a real disk that meets our assumptions, will behave as the ConcreteSystem does, and hence as a CrashSafe⟨Map⟩. We have connected not just code, but a system with a disk and the possibility of the code crashing, up to the app spec.

7 Evaluation

Our evaluation addresses two main questions:

1. Do our automation-control techniques (§4) and language improvements (§5) improve the developer experience?
2. Can our verification methodology scale to the complexity of a modern key-value-store data structure, and can we deliver the performance gains of write optimization?

7.1 Developer Experience

Measuring Tedium. We estimate the amount of tedium (or conversely, the efficacy of automation) by the ratio of the lines of proof (e.g., lemmas, pre-/post-conditions, loop invariants) to the lines of executable implementation code. This is not a precise model, since it measures a completely verified artifact, where the developers may have cleaned up temporary lines of tedium that were typed in the course of resolving verification failures. However, the proof text in the "cleaned up" code at least reflects the tedium needed to bridge what automation could not manage by itself.

Table 1 gives line counts for VeriBetrKV, organized by the major components shown in Figure 3. We see that the proof ratio for the implementation code is 4:1, which grows to 7:1 when including all system refinement proofs ("total"). This is comparable to the distributed, in-memory key-value store verified in previous work [29], which also reports a 7:1 ratio. However, VeriBetrKV's implementation is 3× larger, indicating that automated verification techniques can scale to larger systems without super-linear effort.

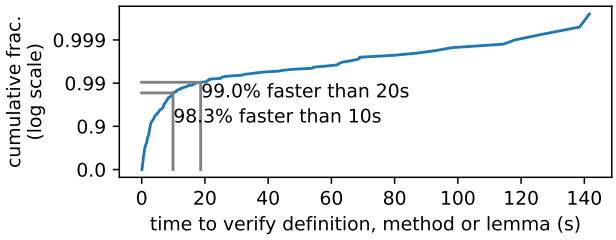


Figure 4: Cumulative distribution of verification times of function definitions, implementation methods, and proof lemmas. Most definitions–99.0%–verify in less than 20s, and 98.3%–verify in less than 10s.

The results also show that VeriBetrKV's implementation is more than 5× larger than its specification, giving us reason to hope that the specification is less likely to contain bugs than an unverified implementation. We have observed no correctness or data loss bugs at runtime; we have seen liveness and performance bugs.

Table 2 compares two VeriBetrKV components that we wrote using both dynamic frames and linear reasoning. The results show that switching to linear reasoning saves tedium, reducing proof overhead by 31–37%. Our qualitative experience was that development of linear code was much more pleasant than dynamic frames because the linear typechecker quickly and unambiguously identifies aliasing problems.

One interesting datum for tedium is the effort developers spend on test infrastructure in conventional development. RocksDB has a 0.99:1 ratio between test and production code (measuring its `db`, `java`, `utilities`, `third-party`, and `tools` directories). BerkeleyDB's ratio is 0.45.

Measuring Proof-Attempt Latency. To assess the success of our discipline for keeping automation under control (§4), we measure the time to verify individual proof units (e.g., definitions, methods, or lemmas) where developers spend most of their time waiting for verification results. This estimates the developer's perception of the latency of the verification-development cycle.

Figure 4 demonstrates that VeriBetrKV almost always exhibits interactive verification times, with 98.3% of definitions verifying in under 10 seconds, and 99.0% in under 20 seconds. This suggests that our timeout-averse development policy is

[3] The implementation of the hash table (with dynamic frames and the linear type system) and search tree (with dynamic frames) coincidentally have identical line counts, despite being unrelated.

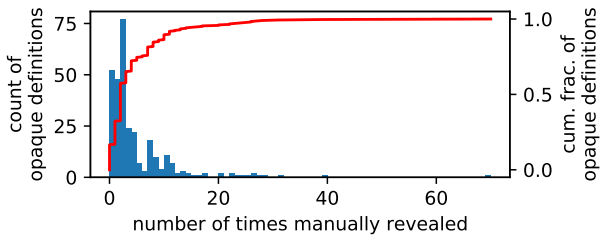


Figure 5: VeriBetrKV manually hides 308 definitions (18% of all system definitions), revealing them 1403 times. This metric reflects the effort involved in explicitly managing automation.

effective. The figure reveals that we did not apply our policy with perfect consistency, as 37 proofs do take longer than 20s to verify. Of those, 54% involve dynamic frames.

It is difficult to capture every place we applied the policy. As a proxy, we can count places where we explicitly hid a definition. This both overestimates the cost of the policy (because sometimes we hid a definition based on intuition, before observing a timeout) and underestimates it (because automation can be controlled with other techniques).

These approximations aside, Figure 5 gives an idea of the burden of controlling automation. We hid 308 definitions, 18% of all definitions in the system. These hidden definitions are manually revealed 1403 times. Of the 308, 52 were never revealed: the salient properties of the definition could be exported as a postcondition without causing timeouts. 74% of hidden definitions are revealed no more than five times; their essential features are captured in lemmas or wrapped into higher-level definitions.

One of the motivations for introducing linear types into Dafny (§5) is to remedy slow verification of dynamic frames. For the fragment of VeriBetrKV we converted to linear reasoning, we compared interactive verification times (as in Figure 4) against the original dynamic frames code. The maximum method-level interactive verification time dropped 10s to 32s, and the 99th percentile dropped 1.3s to 4.8s.

Developers are typically less sensitive to the latency of continuous-integration builds that check that the system as a whole still verifies. For VeriBetrKV, these take 1.8 hours of CPU time, but thanks to the inherent parallelism of modular verification, complete in 11 minutes on 32 cloud machines.

7.2 Performance

Our performance evaluation addresses two questions:

1. Does VeriBetrKV demonstrate the insertion-performance gains of write-optimized data structures?
2. Does our linear extension produce code with performance comparable to hand-written code using dynamic frames?

All experiments were run on cloud instances with directly attached physical storage. The HDD experiments and subcomponent microbenchmarks are run on AWS EC2 d2.xlarge instances, with 4x hardware hyperthreads on a 2.4 GHz Intel Xeon E5-2676 v3. The SSD experiments are run on AWS EC2 i2.xlarge instances, with 4x hardware hyperthreads on a 2.5 GHz Intel E5-2670 v2 and a SATA SSD.

7.2.1 YCSB

Figure 6 shows throughput for VeriBetrKV, BerkeleyDB, and RocksDB on the YCSB benchmarks [14] on hard drive and SSD, including the load phase for workload A, and a uniformly random query workload (labeled as workload "U"). All systems are limited to a single core.

There are three main take-aways from these measurements. First, VeriBetrKV demonstrates the performance gains of using a write-optimized data structure. For the load phase on hard drive, which consists of a pure random insertion workload, VeriBetrKV is over 25× faster than BerkeleyDB. Even on SSD, where random I/O is much cheaper, VeriBetrKV modestly outperforms BerkeleyDB on insertions.

In contrast, VeriBetrKV is roughly 8× slower than RocksDB on hard disk and 4× slower on SSD. Investigation identified three contributing factors: First, where Rocks relies on the kernel buffer cache, VeriBetrKV manages its own cache memory. Its effective cache size is reduced due to malloc fragmentation and conservative allocation to avoid violating the cgroup. Simulating an efficiently-allocated 1.8GiB B$^\varepsilon$tree node cache improves performance to over 13K insertions per second on HDD. Second, VeriBetrKV fails to overlap CPU with flush I/O: a twelve-minute run spends four minutes in CPU and eight minutes waiting for I/O, accounting for roughly a $2/3\times$ slowdown. Third, VeriBetrKV performs 1.5× more I/O than Rocks in YCSB-Load; about half of the extra I/O is due to a suboptimal checkpointing policy.

The second main take-away is that queries in VeriBetrKV are about 4× slower than on RocksDB. The fragmentation penalty again explains a 2× factor; with a simulated 1.8GiB cache, VeriBetrKV and Rocks both perform 300K I/Os in serving 1M YCSB-C queries. Furthermore, we observed most Rocks I/Os are 4-8KiB (one or two buffer cache pages), whereas most VeriBetrKV I/Os are 1.5MiB (an entire B$^\varepsilon$tree node). This I/O size difference combined with the measured seek and read bandwidth of our hard drives explains the remaining gap. We plan to change our marshalling strategy to enable reading fields without fetching the entire node.

The final take-away is that at a macro-level our linear implementation has essentially the same performance as the version with hand-tuned code using dynamic frames reasoning.

Overall, we conclude that VeriBetrKV demonstrates that a verified system can achieve the performance gains of a write-optimized storage system, but it needs further optimization to match highly-tuned commercial implementations.

7.2.2 Linear Data Structures

Figure 7 shows the performance of our linear and dynamic-frames-based hash-table and search-tree implementations. The main take-away from both experiments is that, even in mi-

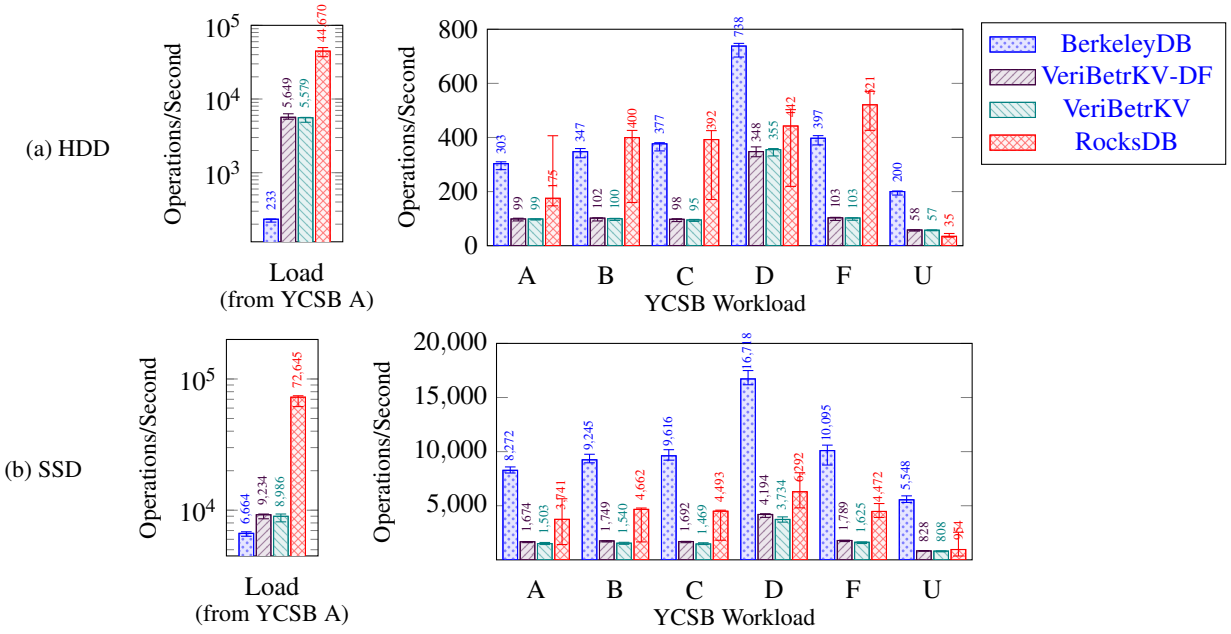
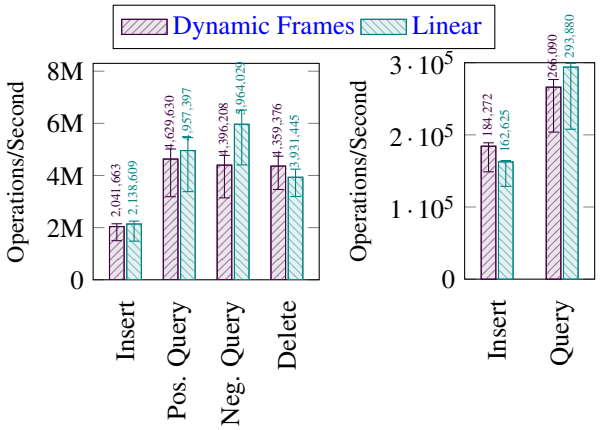


Figure 6: Median throughput of YCSB workloads values running on an HDD (a) and SSD (b) and 2GiB of RAM. VeriBetrKV-DF is a version of our system with hand-tuned code using dynamic frames reasoning. Load is 10M operations (≈ 5GiB of data) and runs are 10000 operations each. Error bars indicate min/max of 6 runs. Higher is better. On an HDD (a), VeriBetrKV insertions are over 25× faster than in BerkeleyDB, but lag RocksDB, by about 9×. VeriBetrKV queries are are about 4× slower than both. On an SSD (b), VeriBetrKV still beats BerkleyDB on random insertions, but VeriBetrKV queries are slower than BerkeleyDB.



(a) Hash table (b) Search tree

Figure 7: Median throughput of subcomponent microbenchmarks. Higher is better. Error bars are min/max of 6 runs.

crobenchmarks, the linear and non-linear implementations have very comparable performance.

The hash table benchmark inserts 64 million key-value pairs, performs 64 million positive and 64 million negative queries, and then deletes everything in the hash table. The keys are selected pseudo-randomly and are distributed uniformly in the 64-bit key space. The linear version is slightly faster than the non-linear version, except for deletes, which are slightly slower. We suspect the speedup comes primarily from the lack of shared pointer overhead.

The search-tree benchmark measures the time to insert 8 million key-value pairs in pseudo-random order and then query them all in the same order. Performance for the linear version is close to the non-linear version, but slightly faster for queries and slightly slower for inserts. We believe queries are faster due to the elimination of shared pointer overheads, and the insertions are slower due to the overheads of destructing nodes on the way down the tree and reconstructing them on the way back up.

Our modifications to add linear types to Dafny consist of 1900 lines (3%); the C++ backend changes, which include linear features, add another 3100 lines (7.5%).

Overall, our linear type system enables us to construct performant code without the challenges of dynamic frame reasoning.

8 Related Work

IronFleet [29], VeriBetrKV's most direct intellectual ancestor, verifies distributed systems of fail-stop nodes. Its verification strategy uses a refinement hierarchy with just two layers: one from imperative code to a protocol state machine, another from there to application spec. VeriBetrKV's implementation contains more components (an in-memory B-tree and hash ta-

ble, a block cache, and a journal), and the B$^\epsilon$tree, its core data structure, is substantially more complicated. Hence VeriBetrKV's implementation is 3× larger than IronFleet's sharded distributed key-value in-memory-only store.

8.1 Verified Storage Systems

FSCQ's Crash Hoare Logic[12] modifies Hoare logic to explicitly reason about crashes, enabling crash reasoning to follow Hoare clauses up the implementation call graph. It does not employ refinement reasoning. FSCQ reasons about potentially repeated crashes during the recovery procedure; VeriBetrKV avoids such reasoning by virtue of a design whose recovery procedure requires no disk writes. FSCQ's implementation is functional code extracted to Haskell, so the framework gives the developer less low-level control than VeriBetrKV.

DFSCQ [11] contributes an application spec for how crashes interplay with the separate `fsync` and `fdatasync` operations. Production file systems like ext4 provide these operations to offer greater performance at the cost of an application spec even more relaxed than the general `sync` operation that covers all updates (as in VeriBetrKV). DFSCQ's implementation exploits this freedom to defer writes.

Yggdrasil uses refinement to show implementation functional correctness relative to a specification [54]. Crashes in the environment are implicit, and the app spec only promises linearity, requiring the implementation to sync on every mutating client operation (or group commit). It cannot exploit the performance benefit of deferring writes until an application-specified `sync`. Its disk model includes asynchronous I/O but has no concept analogous to an IOSystem. Its pushbutton approach to verification constrains the structure of the implementation and proof, but in exchange produces a very favorable proof:code ratio, reported at 1:300.

As discussed in §5, our linear extensions to Dafny are inspired by multiple sources [2, 33, 36, 51, 59]. Most relevant to systems verification is the Cogent language [2], which is a restricted functional language that certifiably compiles to C code. Amani et al. use Cogent to develop two file systems, each about 4K lines of native C. They verify two functional correctness properties of one of the file systems, with the aim of eventually proving both functional correctness and crash safety. The Cogent language is an impressive foundational effort and its certifying compiler provides a stronger guarantee than our simple but trusted changes to Dafny's type system. Our type system's linear variables are similar in spirit to Cogent's linear types, but our work also integrates linearity directly with Dafny's dynamic frames, enabling us to move smoothly back and forth between linear and non-linear reasoning about memory.

It is difficult to make meaningful performance comparisons between our durable key-value store and file systems: File systems provide richer semantics, such as bulk directory rename.

8.1.1 Concurrent Storage

AtomFS [66] is a compute-concurrent file system with fine-grained (per-inode) locks, but it does not reason about crash safety. CSPEC [9] verifies a compute-concurrent mail server absent crash safety. It verifies 215 lines of Coq implementation with 4,050 lines of proof. Perennial [10] verifies a compute-concurrent, crash-safe mail server. Perennial extends a capability separation logic with crash-safety-specific concepts, with which it builds a refinement argument. Perennial verifies 159 lines of concurrent Go with 3,360 lines of proof. Drawing on techniques from these systems would allow VeriBetrKV to scale further via parallelism.

8.2 Automation Strategies

A line of work on "push-button" verification [45, 46, 54, 55, 64, 65] accepts constraints on system structure in exchange for maximizing automation. The developer constrains their imperative code to bounded executions and writes invariants to span independent handler invocations. Such handlers could not walk down a tree of arbitrary depth, as happens in VeriBetrKV's B$^\epsilon$tree and search tree. Supporting longer bounded executions requires framework improvements [45] rather than creating a modularization job for the developer.

Taube et al. [57] employ a restricted fragment of logic [49] to verify distributed system implementations, including Raft [48] and Multi-Paxos [39]. By restricting the description of the protocol and its properties to a decidable logic, this approach guarantees that a solver can always return either a decisive answer. While the developer still must supply invariants, the remaining proof work is entirely automatic. The cost of this approach is that it force the developer to restate definitions unnaturally, and decidable verification is still subject to combinatorial slowdown as the scope of definitions grows.

The exploration of extreme points in the automation space is promising, but limitations on expressiveness and design motivate us to stick with developer-guided proofs, and instead use automation to make it as cheap as possible.

8.3 Additional Verified Systems

The seL4 verified microkernel is the seminal systems verification project [34], demonstrating the feasibility of verifying software at the scale of thousands of lines. C code refines a Haskell functional model of the implementation, which refines a high-level specification for the behavior of system calls.

CertiKOS [27] proves a concurrent microkernel implementation correct using refinement of state machines it calls "layers" expressed in a side-effect free subset of C. It introduces the notion of contextual refinement to reason about concurrent state machines in isolation [28].

Verdi [61] uses Coq to verify distributed systems by proving the correctness of a series of "system transformers" that take a high-level protocol description running in a friendly environment and transform it into a protocol that is safe in a more hostile environment (e.g., where packets can be dropped). The

signature transformer is a verified implementation of Raft [62]. In a sense, Verdi is a distributed systems analog to correctness-preserving compiler transforms.

9 Conclusion

In this work, we extracted a general methodology for verifying asynchronous systems from prior work and applied it to storage systems. In doing so, we developed a verification discipline and a novel integration of linearity with dynamic frame reasoning to reduce the burden of verifying systems code. Because we applied a generic methodology, we expect these improvements to apply equally well to the verification of other asynchronous systems. In future work, we would like to extend the methodology to also support thread-concurrent systems with shared memory, utilizing our linear type system to manage memory ownership.

Ultimately, our implementation and proof of crash safety for VeriBetrKV, a complex, modern storage system, show that automated verification techniques can scale to larger code bases without increasing the proof burden relative to simpler systems.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Gernot Heiser, for useful feedback on the paper. Work at CMU was supported, in part, by grants from a Google Faculty Fellowship, the Alfred P. Sloan Foundation, and the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521. Andrea Lattuada is supported by a Google PhD Fellowship.

A Artifact Appendix

A.1 Abstract

The evaluated artifact is provided as Docker images that contain the source code to VeriBetrKV, build instructions to run the verification as well as run the performance experiments and draw the graphs corresponding to those in §7 with the generated data.

A.2 Artifact check-list

- **Algorithm**: A verified B$^\epsilon$tree-based key-value store.
- **Program**: VeriBetrKV as described in this paper.
- **Compilation**: Dafny/C++ compiler, included in Docker image.
- **Data set**: YCSB generated workload
- **Run-time environment**: Docker
- **Hardware**: Any x64; provide a data store directory on HDD or SSD as desired
- **Run-time state**: KV store backing files
- **Output**: PDFs containing graphs corresponding to §7
- **Required disk space**: 20GiB
- **Expected experiment run time**: Several hours
- **Public link**: https://github.com/secure-foundations/veribetrkv-osdi2020/blob/master/README.md

A.3 Description

A.3.1 How to access

Follow the README at https://github.com/secure-foundations/veribetrkv-osdi2020/blob/master/README.md. You can either run the binary Docker distribution, or build it yourself.

A.3.2 Hardware dependencies

You will need any x86 CPU, plus HDD and/or SSD storage devices for the performance measurements.

A.3.3 Software dependencies

All required dependencies are included in the Docker image.

A.3.4 Data sets

Performance experiments use the YCSB benchmark, for which the source and configuration are included in the Docker image.

A.4 Installation

You can either download the GitHub release, `veribetrkv-artifact-hdd`, and load the image with

```
docker load -i veribetrkv-artifact-hdd.tgz
```

or build it yourself with

```
cd docker-hdd
docker build -t veribetrkv-artifact-hdd .
```

A.5 Experiment workflow

The README explains how to launch the experiments by running scripts from outside Docker. The scripts will generate PDFs that reproduce the results from the paper.

A.6 Evaluation and expected result

The graphs in the output PDFs should correspond to those in §7, modulo variation in the experimental hardware.

A.7 Experiment customization

The README at the link above provides details on how to modify the experiment scripts in the Docker container.

A.8 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts

References

[1] Abadi, M., and Lamport, L. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (May 1991).

[2] Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., and Heiser, G. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[3] ASTRAUSKAS, V., MÜLLER, P., POLI, F., AND SUMMERS, A. J. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2019).

[4] BENDER, M., PANDEY, P., PORTER, D., YUAN, J., ZHAN, Y., CONWAY, A., FARACH-COLTON, M., JANNEN, W., JIAO, Y., JOHNSON, R., KNORR, E., MCALLISTER, S., AND MUKHERJEE, N. Small refinements to the DAM can have big consequences for data-structure design. In *Proceeding of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2019).

[5] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND ZHAN, Y. An introduction to B^ϵ-trees and write-optimization. *;login: The USENIX Magazine 40*, 5 (Oct. 2015), 22–28.

[6] BÖHME, S., AND WEBER, T. Fast LCF-style proof reconstruction for Z3. In *Proceedings of Interactive Theorem Proving* (2010).

[7] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (Baltimore, MD, USA, 2002), pp. 39–48.

[8] CASTAGNOLI, G., BRAUER, S., AND HERRMANN, M. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. *IEEE Transactions on Communications 41*, 6 (1993), 883–892.

[9] CHAJED, T., KAASHOEK, M. F., LAMPSON, B., AND ZELDOVICH, N. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2018).

[10] CHAJED, T., TASSAROTTI, J., KAASHOEK, M. F., AND ZELDOVICH, N. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)* (Hunstville, ON, Canada, Oct. 2019).

[11] CHEN, H., CHAJED, T., KONRADI, A., WANG, S., ILERI, A., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (Shanghai, China, Oct. 2017).

[12] CHEN, H., ZIEGLER, D., CHAJED, T., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP 2015)* (Monterey, California, Oct. 2015).

[13] COHEN, E., DAHLWEID, M., HILLEBRAND, M., LEINENBACH, D., MOSKAL, M., SANTEN, T., SCHULTE, W., AND TOBIES, S. VCC: A practical system for verifying concurrent C. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics* (2009).

[14] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010).

[15] COQ DEVELOPMENT TEAM. The Coq Proof Assistant https://coq.inria.fr/.

[16] DE MOURA, L. M., AND BJØRNER, N. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[17] DESAI, A., GUPTA, V., JACKSON, E., QADEER, S., RAJAMANI, S., AND ZUFFEREY, D. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013).

[18] DRĂGOI, C., HENZINGER, T. A., AND ZUFFEREY, D. Psync: A partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2016).

[19] FACEBOOK, INC. MyRocks: A RocksDB Storage Engine with MySQL. http://myrocks.io/.

[20] FACEBOOK, INC. RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/.

[21] FERRAIUOLO, A., BAUMANN, A., HAWBLITZEL, C., AND PARNO, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2017).

[22] FISHER, K., LAUNCHBURY, J., AND RICHARDS, R. The HACMS program: Using formal methods to eliminate exploitable bugs. *Philosophical Transactions A, Math Phys Eng Sci. 375*, 2104 (Sept. 2017).

[23] FLOYD, R. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics* (1967).

[24] FONSECA, P., KAIYUAN ZHANG, X. W., AND KRISHNAMURTHY, A. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of ACM EuroSys Conference* (Apr. 2017).

[25] GARLAND, S. J., AND LYNCH, N. A. Using I/O automata for developing distributed systems. *Foundations of Component-Based Systems 13* (2000).

[26] GOOGLE, INC. LevelDB. https://github.com/google/leveldb.

[27] GU, R., KOENIG, J., RAMANANANDRO, T., SHAO, Z., WU, X. N., WENG, S.-C., ZHANG, H., AND GUO, Y. Deep specifications and certified abstraction layers. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (2015).

[28] GU, R., SHAO, Z., CHEN, H., WU, X., KIM, J., SJÖBERG, V., AND COSTANZO, D. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016).

[29] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2015).

[30] HAWBLITZEL, C., HOWELL, J., LORCH, J. R., NARAYAN, A., PARNO, B., ZHANG, D., AND ZILL, B. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2014).

[31] HOARE, T. An axiomatic basis for computer programming. *Communications of the ACM 12* (1969).

[32] KASSIOS, I. T. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods* (2006).

[33] KLABNIK, S., NICHOLS, C., AND RUST COMMUNITY. The Rust Programming Language https://doc.rust-lang.org/book/.

[34] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., MURRAY, T., SEWELL, T., KOLANSKI, R., AND HEISER, G. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems 32*, 1 (2014).

[35] KUMAR, R., MYREEN, M. O., NORRISH, M., AND OWENS, S. CakeML: a verified implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (Jan. 2014).

[36] LAHIRI, S. K., QADEER, S., AND WALKER, D. Linear maps. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification* (2011).

[37] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[38] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994).

[39] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[40] LAMPORT, L. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[41] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2010).

[42] LEROY, X. Formal verification of a realistic compiler. *Communications of the ACM (CACM) 52*, 7 (2009), 107–115.

[43] MONGODB, INC. The WiredTiger Storage Engine. http://www.wiredtiger.com/.

[44] MULLEN, E., PERNSTEINER, S., WILCOX, J. R., TATLOCK, Z., AND GROSSMAN, D. Œuf: Minimizing the Coq extraction TCB. In *Proceedings of the ACM Conference on Certified Programs and Proofs (CPP)* (2018).

[45] NELSON, L., BORNHOLT, J., GU, R., BAUMANN, A., TORLAK, E., AND WANG, X. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[46] NELSON, L., SIGURBJARNARSON, H., ZHANG, K., JOHNSON, D., BORNHOLT, J., TORLAK, E., AND WANG, X. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[47] NIPKOW, T., PAULSON, L., AND WENZEL, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[48] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (June 2014).

[49] PADON, O., MCMILLAN, K. L., SAGIV, M., PANDA, A., AND SHOHAM, S. Ivy: Safety verification by interactive generalization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2016).

[50] PERCONA LLC. The PerconaFT Storage Engine. https://github.com/percona/PerconaFT.

[51] QADEER, S., TASIRAN, S., AND HAWBLITZEL, C. Automated and modular refinement reasoning for concurrent programs. In *Computer Aided Verification (CAV)* (2015).

[52] REYNOLDS, J. C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (2002).

[53] SCYLLA, INC. ScyllaDB: The real-time big data database. https://www.scylladb.com.

[54] SIGURBJARNARSON, H., BORNHOLT, J., TORLAK, E., AND WANG, X. Push-button verification of file systems via crash refinement. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2016).

[55] SIGURBJARNARSON, H., NELSON, L., CASTRO-KARNEY, B., BORNHOLT, J., TORLAK, E., AND WANG, X. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2018).

[56] SWAMY, N., HRIŢCU, C., KELLER, C., RASTOGI, A., DELIGNAT-LAVAUD, A., FOREST, S., BHARGAVAN, K., FOURNET, C., STRUB, P.-Y., KOHLWEISS, M., ZINZINDOHOUÉ, J.-K., AND ZANELLA-BÉGUELIN, S. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages* (2016).

[57] TAUBE, M., LOSA, G., MCMILLAN, K. L., PADON, O., SAGIV, M., SHOHAM, S., WILCOX, J. R., AND WOOS, D. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2018).

[58] V. GLEISSENTHALL, K., KICI, R. G., BAKST, A., STEFAN, D., AND JHALA, R. Pretend synchrony: Synchronous verification of asynchronous distributed programs. vol. 3, Association for Computing Machinery.

[59] WADLER, P. Linear types can change the world! In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods* (1990).

[60] WHEELER, D. A. SLOCCount. Software distribution. http://www.dwheeler.com/sloccount/.

[61] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2015).

[62] WOOS, D., WILCOX, J. R., ANTON, S., TATLOCK, Z., ERNST, M. D., AND ANDERSON, T. Planning for change in a formal verification of the raft consensus protocol. In *ACM Conference on Certified Programs and Proofs (CPP)* (Jan. 2016).

[63] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *PLDI* (June 2011).

[64] ZAOSTROVNYKH, A., PIRELLI, S., IYER, R., RIZZO, M., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[65] ZHANG, K., ZHUO, D., AKELLA, A., KRISHNAMURTHY, A., AND WANG, X. Automated verification of customizable middlebox properties with Gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (2020).

[66] ZOU, M., DING, H., DU, D., FU, M., GU, R., AND CHEN, H. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)* (Hunstville, ON, Canada, Oct. 2019).



Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache

Xingda Wei, Rong Chen, Haibo Chen
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

RDMA (Remote Direct Memory Access) has gained considerable interests in network-attached in-memory key-value stores. However, traversing the remote tree-based index in ordered stores with RDMA becomes a critical obstacle, causing an order-of-magnitude slowdown and limited scalability due to multiple roundtrips. Using index cache with conventional wisdom—caching partial data and traversing them locally—usually leads to limited effect because of unavoidable capacity misses, massive random accesses, and costly cache invalidations.

We argue that the machine learning (ML) model is a perfect cache structure for the tree-based index, termed *learned cache*. Based on it, we design and implement XSTORE, an RDMA-based ordered key-value store with a new hybrid architecture that retains a tree-based index at the server to perform dynamic workloads (e.g., inserts) and leverages a learned cache at the client to perform static workloads (e.g., gets and scans). The key idea is to decouple ML model retraining from index updating by maintaining a layer of indirection from logical to actual positions of key-value pairs. It allows a stale learned cache to continue predicting a correct position for a lookup key. XSTORE ensures correctness using a validation mechanism with a fallback path and further uses speculative execution to minimize the cost of cache misses. Evaluations with YCSB benchmarks and production workloads show that a single XSTORE server can achieve over 80 million read-only requests per second. This number outperforms state-of-the-art RDMA-based ordered key-value stores (namely, DrTM-Tree, Cell, and eRPC+Masstree) by up to 5.9× (from 3.7×). For workloads with inserts, XSTORE still provides up to 3.5× (from 2.7×) throughput speedup, achieving 53M reqs/s. The learned cache can also reduces client-side memory usage and further provides an efficient memory-performance tradeoff, e.g., saving 99% memory at the cost of 20% peak throughput.

1 Introduction

Network-attached in-memory key-value stores have become the foundation of many datacenter applications, including databases [47, 55], distributed file systems [7], web services [4, 37], and serverless computing [23, 42, 28], to name a few. With the prevalence of affordable high-performance networks in modern datacenters [46, 17, 20], such as InfiniBand, RoCE, or OmniPath, CPU quickly becomes the performance bottleneck and limits the scalability with the increase of clients [31]. RDMA (Remote Direct Memory Access) has recently generated considerable interests in optimizing network-attached in-memory key-value stores (aka RDMA-based KVs) in both academia [34, 25, 52] and industry [16, 55, 31], as it enables direct access to the memory of remote machines with low latency and CPU/kernel bypassing. However, leveraging RDMA to ordered key-value stores encounters a significant obstacle—traversing tree-based index with one-sided RDMA primitives is costly and complex (e.g., 11× slowdown in Fig. 2c). This is because it usually requires multiple network round trips (e.g., $O(\log N)$) and rapidly saturates bandwidth.

Many recent academic and industrial efforts [57, 17, 35] therefore proposed *index caching* to reduce RDMA operations. Yet, the conventional wisdom on implementing cache—replicating partial data and accessing them locally—does not work well with the tree-based index, and the drawbacks are amplified by maintaining the *tree-based* cache with RDMA primitives. First, the tree-based index can be large, so that the cache would suffer from unavoidable capacity misses. Second, the cache would aggravate random memory accesses and further increase the end-to-end latency. Third, updating the tree-based index may recursively invalidate the cache and cause false invalidation due to path sharing.

Inspired by recent research [29]—using machine learning (ML) models as an alternative index structure, we propose to leverage ML models as the (client-side) RDMA-based cache for the (server-side) tree-based index, termed *learned cache*. Specifically, the client uses learned cache to predict a small range of positions for a lookup key and then fetches them using one RDMA READ. After that, the client uses a local search (e.g., scanning) to find the actual position and fetches the value using another RDMA READ. Although using ML models as the index seems efficient (a few floating/int operations) and cheap (a small memory footprint) for static workloads (e.g., gets), it is also notoriously slow (frequently retraining ML models) and costly (keeping data in order) for dynamic workloads (e.g., inserts).

To address the above challenges, we propose a *hybrid* architecture that retains a tree-based index at the server to perform dynamic workloads (e.g., inserts) and leverages a learned cache at the client to perform static workloads (e.g., gets and scans). The hybrid architecture not only provides separate and appropriate execution paths for both workloads, but also simplifies the mechanism to guarantee the correctness of concurrent local and remote operations.

Based on this architecture, we further introduce a layer of indirection (i.e., a translation table) between the ML model and the tree-based index, which maps the logical position to the actual position of key-value pairs in the leaf-node granularity. The translation table decouples model retraining from index updating (e.g., node splits) and allows a *stale* learned cache (a combination of ML model and translation table) to continue predicting a *correct* position for a lookup key, as long as it is not overlapped with a leaf node split. It implies that the tree-based index can be concurrently updated in-place. Meanwhile, the ML model associated with its translation table can be retrained in the background and independently pulled by the clients on demand.

We have implemented XSTORE by extending a concurrent B+tree [50] with a well-tuned RDMA framework [51]. We evaluate XSTORE using the YCSB benchmarks [13] with two synthetic and one real-world [2] datasets, as well as two production workloads from Nutanix [30]. Our experimental results show that a single XSTORE server can achieve over 80 million read-only requests per second. This number outperforms state-of-the-art RDMA-based ordered key-value stores (i.e., DrTM-Tree [11], Cell [35], and eRPC+Masstree [24]) by up to 5.9× (from 3.7×). For workloads with inserts, XSTORE still provides up to 3.5× (from 2.7×) throughput speedup, achieving 53M reqs/s. The learned cache also reduces client-side memory usage significantly and further provides an efficient memory-performance tradeoff. For example, it can save 99% memory at the cost of 20% peak throughput, compared to caching the whole index.

In summary, this paper makes four contributions:

- The idea of *learned cache* that leverages machine learning (ML) models as index cache for RDMA-based, tree-backed KV stores;
- A hybrid architecture that combines (client-side) learned cache and (server-side) tree-based index to embrace static and dynamic workloads;
- A layer of indirection (translation table) that decouples ML model retraining from index updating and allows a *stale* learned cache to predict a *correct* position;
- A prototype implementation and an evaluation that demonstrates the advantage and efficacy of XSTORE.

2 RDMA-based Key-Value Store

In this paper, we focus on in-memory key-value (KV) stores that adopt the client-server model (network-attached) [34, 32, 25, 8] and range index structures (tree-backed) [33, 35, 57]. The server hosts both key-value pairs and indexes in main memory and handles requests from multiple clients concurrently. The client interacts with the server through a library that provides basic key-value interfaces, including GET(K), UPDATE(K,V), SCAN(K,N)[1], INSERT(K,V), and DELETE(K), as well as more complex operations built atop them.

[1] SCAN(K,N) provides a form of range query that retrieves first (up to) N key-value pairs, where their keys are larger than or equal to K.

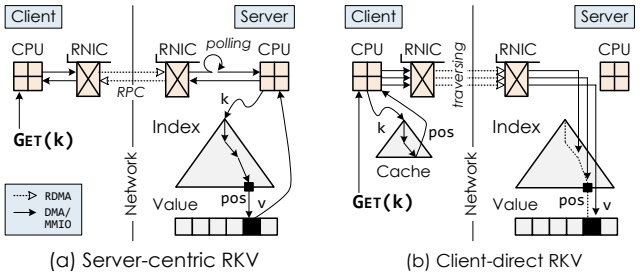


Fig. 1. The architecture of RDMA-based key-value stores: (a) server-centric RKV and (b) client-direct RKV.

RDMA (Remote Direct Memory Access) is an emerging feature—appearing in affordable high-performance networks (e.g., InfiniBand, RoCE, or OmniPath)—that enables direct access to the memory of remote machines with low latency and CPU/kernel bypassing. It has generated considerable interest in deploying the network in modern datacenters [17, 46, 20] and optimizing key-value stores (aka RDMA-based KVs) [34, 25, 16, 9, 8]. However, few prior systems consider *ordered* key-value stores that rely on tree-based indexes to handle range queries (i.e., SCAN(K,N)).

Server-centric design (S-RKV) [52, 26, 24]. An obvious design is to take a traditional KV store and reimplement the communication layer (e.g., RPC) using RDMA primitives. As shown in Fig. 1a, the clients ship their requests to the server via RDMA network using one round trip for each; the server traverses the tree-based index and performs the request locally. The *server-centric* design allows access to the server-side store with only two RDMA operations (one for sending and one for receiving), no matter how complex the index structures are, thereby avoiding multiple round trips and message size amplification [26]. However, this design exploits only high performance (low latency and high bandwidth) but not CPU efficiency (remote CPU bypassing) of RDMA network at the server, which limits the scalability of these KV stores with the increase of clients.

Client-direct design (C-RKV) [35, 17, 57]. The adoption of RDMA makes it practical to allow clients to access data hosted on the server directly, thereby permitting an alternative (*client-direct*) design that relaxes the burden on server CPUs. To simplify the mechanism for consistency, this design is restricted to read-only requests (i.e., GET and SCAN) in most systems [34, 16, 35]. This common choice is also motivated by the read-dominated nature of most applications [6]. As shown in Fig. 1b, the clients use one-sided RDMA operations to traverse the tree-based index and fetch the value directly for read-only requests; the server still needs to perform the rest of requests (i.e., UPDATE, INSERT, and DELETE) locally. The *client-direct* design can shift the CPU load on the server to the clients, which would alleviate the bottleneck (from CPU to network), especially on high-bandwidth networks (e.g., 100Gbps). However, it may consume extra network round trips for traversing the tree-based index due to the lack of richness of RDMA primitives, causing an order-

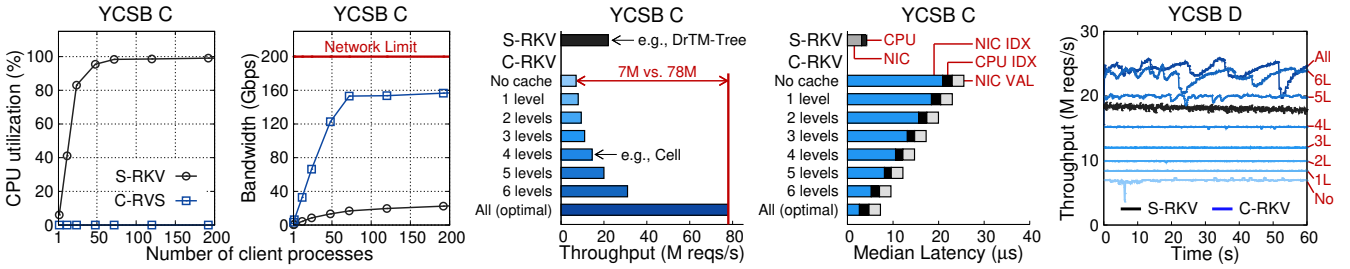


Fig. 2. A comparison of server-side (a) CPU and (b) network bandwidth utilization, (c) peak throughput, (d) end-to-end median latency at low load, and (e) throughput timeline for state-of-the-art server-centric (**S-RKV**) and client-direct (**C-RKV**) RDMA-based KV stores. **Workload**: YCSB C (100% read) and YCSB D (95% read and 5% insert), using 100M keys with a uniform distribution. **Testbed**: The server has two 12-core CPUs and two 100Gbps RNICs.

of-magnitude slowdown (e.g., 11× slowdown in Fig. 2c). For example, recent work [35, 57] uses RDMA READs to traverse remote B+tree index and invariably incurs multiple network round trips (O(logN) [3]).

Recently, *index caching* has been proposed to reduce network round trips for index traversal by RDMA-based systems [52, 48, 17, 35, 38], namely, the client caches the server-side index locally. It aims at reducing RDMA READs for fetching the position of the value (aka *lookup*), instead of caching the value directly.[2] Thus, an *optimal* result with index caching only needs two RDMA operations per request (one for lookup and one for read).

3 Analysis of RDMA-based Ordered KVs

CPU is the primary scalability bottleneck in the server-centric design. Fig. 2 compares hardware resource utilization between S-RKV and C-RKV with the increase of clients. For S-RKV, the server rapidly saturates all CPUs (24 cores) but just consumes 11% of network bandwidth. It implies that CPU first becomes the performance bottleneck and limits the scalability with the increase of clients, especially when deploying fast networks. This also runs counter to the recent trend of building servers in modern datacenters with CPU-bypassing networks [17, 46, 20]. As shown in Fig. 2c, S-RKV reaches the peak throughput of around 24M reqs/s. Traversing tree-based index occupies most of CPU time, as it involves massive random memory accesses. On our testbed, we measured that one CPU core can perform 43 million 64-byte random reads per second at full speed. Thus, each core can only process up to 1.8M reqs/s for traversing a (8-level) B+tree with 100M keys, even putting other CPU and network costs aside.

Costly RDMA-based traversal is the key obstacle in the client-direct design. C-RKV allows the client to traverse the server-side index directly by using one-sided RDMA READs, which can thoroughly bypass server CPUs (see Fig. 2a). However, RDMA-based index traversal usually requires multiple network round trips (e.g., O(logN) for tree-based index) and saturates the network bandwidth quickly. As shown in Fig. 2c, RDMA-based traversal limits the peak throughput of C-RKV to 7 million requests per second, even much lower than that of S-RKV. Using index caching at the clients can reduce RDMA operations by traversing index nodes locally. On our testbed, the throughput of C-RKV with index caching, similar to state-of-the-art design (Cell [35]), peaks at 14.5M reqs/s, as each request takes 4 RDMA READs (down from 8) for traversal.

Tree is not a proper structure for RDMA-based index cache. To our knowledge, existing RDMA-based index caches use *homogeneous structures* to store *partial* index nodes, similar to the conventional design. For example, each client replicates tree nodes and traverses them locally before accessing the tree-based index hosted on the server [17, 35, 57].

First, the tree-based index can be large [56, 18, 26], and the traversal demands multiple random accesses from the root to the leaf node. Thus, each client can only cache nodes near the root (e.g., top four levels [35]) to minimize thrashing and maximize hits [35, 17]. Yet, the index cache still suffers from *unavoidable capacity misses* (bottom node levels). In Fig. 2c, for a read-only workload, the effect of RDMA-based caching for tree-based index is dominated by inner node levels cached. The *optimal* throughput (a *whole-index* cache) reaches 78M reqs/s using one RDMA READ for each traversal (fetch the position of value), 3.3× better than S-RKV.

Second, traversing tree-based index is a memory-intensive but low-compute operation. The *homogeneous* index cache can just alter the type of memory accesses (i.e., remote and local), instead of reducing the number of memory accesses (O(logN)). Hence, despite the index cache, traversing tree-based index would still incur *massive random accesses* and suffer from CPU cache misses, TLB misses, and RNIC's page translation cache misses. As shown in Fig. 2d, even caching the whole index, the end-to-end latency of C-RKV is still 80% higher than S-RKV, and the CPU cost on index cache (CPU_IDX) occupies close to 30%.

Third, updates to the tree-based index (i.e., inserts and deletes) might propagate the changes from the leaf level to the root node, so that the index updates would probably invalidate the cache *recursively* [57] and cause *false* invalidations

[2] Considering RDMA performance degradation with increasing payload size [25], the client will only cache internal nodes [35, 38] and not directly fetch a batch of keys and (inline) values to avoid bandwidth amplification [35, 3].

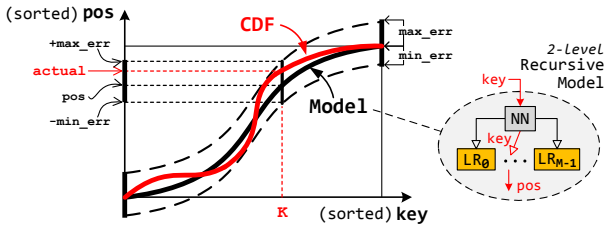


Fig. 3. An example of using ML models to predict the position within a sorted array for a given key.

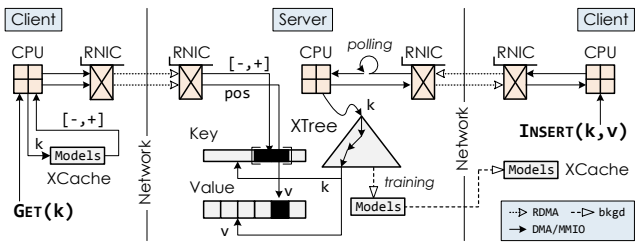


Fig. 4. The hybrid architecture behind XSTORE: client-direct operations (left) and server-centric operations (right).

(path sharing). It would result in frequent cache misses and RDMA READs to retrieve updated index nodes. Worse yet, the more tree nodes cached, the more performance degrades. Further, preserving traversal consistency for dynamic workloads demands sophisticated detection schemes (e.g., fence keys [19, 41]) and incurs additional overhead. In Fig. 2e, the *optimal* throughput significantly drops to 25M reqs/s with severe performance fluctuations, just because of 5% inserts.

4 Approach and Overview

Opportunity: ML Models. Our work is motivated by an attractive observation from the learned index [29]—a range index (e.g., B+tree) that finds the position of a given key inside a sorted array approximates the cumulative distribution function (CDF) of the keys in the index. As shown in Fig. 3, suppose the values have been sorted according to the lookup keys, the CDF (the red curve) is a mapping from the (sorted) keys to the (sorted) positions of their values, namely CDF(K) returns the *actual* position of the value corresponding to K. Prior work [29] proposes to approximate the shape of a CDF using machine learning (ML) models, like neural nets (NN) and linear regression (LR), since they are able to learn a wide variety of distributions. As an alternative range index, the ML model is trained with every key to record the worst over- and under-prediction of a position (i.e., min- and max-error). In Fig. 3, given a lookup key (K), the model (the black curve) can predict a position (pos) with a min- and max-error (min_err and max_err), and a local search (e.g., scanning) around the prediction is used to get the *actual* position. To further reduce the prediction error, a hierarchy of simple models (e.g., recursive-model index [29]) is used to partition the key space, where the model at level L picks the model at level L+1 based on the key.

Our approach: Learned Cache. The key idea behind XSTORE is to leverage machine learning (ML) models as (client-side) RDMA-based cache for the (server-side) tree-based index, termed "***learned cache***". The unique features of machine learning models can fundamentally overcome the drawbacks in the conventional wisdom for RDMA-based index caching (see §3). First, instead of using a *homogeneous* structure to cache a *partial* index, the ML model can cache the *whole* index at the cost of *accuracy*. Therefore, using the learned cache can completely avoid capacity misses, and each lookup only needs one RDMA READ. Further, the ML model is also famously memory-efficient (e.g., two parameters per LR model). Thus, the learned cache can match the *optimal* throughput of conventional design (a whole-index cache) but with practical memory consumption.

Second, instead of finding the *actual* position by traversing a tree-based index with $O(\log N)$ random memory accesses, the ML model can approximately predict a *range* of positions for a lookup key by performing a single multiplication and addition (e.g., linear regression). It implies that the learned cache might also reduce the end-to-end latency, even compared to a whole-index cache, due to fewer CPU cache and TLB misses at the clients.

Finally, instead of *fine-grained* and *recursive* invalidation in the tree-based cache for accurate predictions, the ML model can reduce and delay cache invalidations since it only needs to provide approximate predictions. Updates to the index might only decrease the accuracy of the (partial) ML model. Thus, the learned cache can significantly save invalidation cost in terms of network round trips and bandwidth usage, especially compared to a whole-index cache.

Challenge: Dynamic Workloads. Dynamic workloads (e.g., inserts and deletes) would violate an (unrealistic) assumption of ML-based approach that all key-value pairs are stored in sorted order by key [29]. However, retraining ML models and keeping data in order are slow and costly, which is hard to match the high performance of in-memory key-value stores (tens of millions of requests per second). An intuitive solution is to maintain a delta index (e.g., B+tree) for (in-place or buffer-based) inserts and then periodically compact it with the learned index (data merging and model retraining) [44, 18]. Unfortunately, it cannot work well with RDMA-based index caching. First, additional RDMA-based lookups on the delta index would incur more network round trips and severely increase the latency. Second, it is also hard to cache a fast-changing (tree-based) delta index at the clients. Finally, the data and model compaction definitely interrupts (RDMA-based) remote accesses and completely invalidates the learned cache. Hence, *how to make learned cache keep pace with dynamic workloads at low cost* becomes a key challenge.

Overview of XStore. XSTORE is an in-memory ordered key-value store using a client-server model, where the server and the clients are connected with a high-speed, low-latency net-

work with RDMA.[3] Using ML models as the index (aka *learned index*) is famously efficient and cheap for static workloads (e.g., gets and scans), while it is notoriously slow and costly for dynamic workloads (e.g., inserts and deletes). It is because the inserts would amplify the prediction error and incur model retraining frequently. Prior work [29, 44, 15] relies more on the profit from efficiently handling static workloads to amortize the negative influence on dynamic workloads. We argue that the *learned cache* opens the opportunity to solve this dilemma. Unlike prior work [29, 44, 15, 14], which replaces or augments the tree-based index with the learned index, we propose a *hybrid* architecture that *retains the tree-based index at the server to handle dynamic workloads and uses the learned cache at the clients to handle static workloads*.

The architecture of XSTORE is shown in Fig. 4. The server hosts a B+tree index (XTREE) in the main memory and stores key-value pairs at the leaf level physically, like the common practice. Each client interacts with the server through a library, which hosts a local learned cache (XCACHE). XSTORE uses the client-direct design for read-only requests (i.e., GET(K) and SCAN(K,N)) and the server-centric design for the rest (i.e., UPDATE(K,V), INSERT(K,V), and DELETE(K)). For client-direct operations, like GET(K) in Fig. 4, the client first predicts a range of positions for the key K using XCACHE and then fetches them using one RDMA READ. Finally, the client uses a local search to find the actual position and fetches the value using another RDMA READ. For server-centric operations, like INSERT(K,V) in Fig. 4, the client uses RPC over RDMA to ship the request to the server. The server searches the lookup key K by traversing the B+tree index first and then inserts the new KV pair (K,V). XSTORE will partially retrain ML models for updated tree nodes in the background, and each client will individually fetch the models for XCACHE on demand.

5 Design and Implementation

5.1 Data Structures

XTree. At the server, XSTORE retains a B+tree index (XTREE) and stores key-value pairs at the leaf level physically, like the common practice, as illustrated in the left part of Fig. 5. XTREE follows the basic design of a concurrent B+tree [33, 50], except that the leaf node (LN) adopts the structure optimized for remote reads. The leaf node consists of a 24-bit incarnation (INCA), an 8-bit counter (CNT), a 32-bit right-link pointer to next sibling (NXT), keys with N slots ($K_0 . . K_{N-1}$) and values with N slots ($V_0 . . V_{N-1}$).

Every leaf node is allocated from an RDMA-registered memory region using a slab allocator and can store at most N key-value pairs in sorted order. For brevity, we assume fixed-length key-value pairs here.[4] To save the size of RDMA

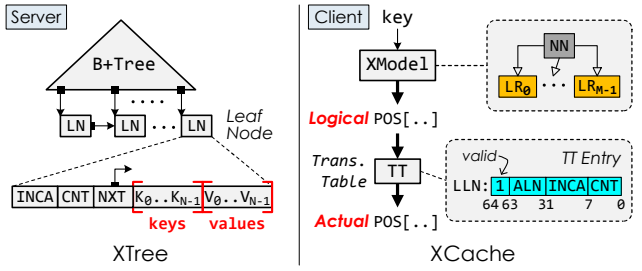


Fig. 5. The main structures in XSTORE: XTREE and XCACHE.

READ for lookup, XSTORE stores keys and values separately but continuously. It can avoid storing the address of the value. The client can fetch N keys from the leaf node and calculate the (remote) address of expected value locally (a fixed offset from its key). Moreover, XSTORE uses incarnation checks [16, 52] to guarantee the consistency of remote accesses. The incarnation in the leaf node is initially zero and is monotonically increased when the leaf node is reused (e.g., split or free). The number of slots (N) can be tuned for RDMA performance (e.g., 16).

XCache. Each client hosts a local learned cache (XCACHE), which consists of a 2-level recursive ML model (XMODEL) and a translation table (TT). As illustrated in the right part of Fig. 5, given a lookup key, XMODEL is used to predict a range of positions (POS[..]) within a sorted array (logically stitching together all leaf nodes of XTREE). Currently, XMODEL uses a linear multi-variate regression model at level 0 (top-model) and simple linear regression models at level 1 (sub-model), a common setup recommended in prior work [15, 29, 44].

The ML model demands the positions (virtual address) of leaf nodes are always sorted by the keys. It is almost impossible for dynamic workloads, since the insertion of key-value pairs may insert a new node at the leaf level and break the sorted order of leaf nodes. The server maintains an additional translation table (TT) for leaf nodes, from logical to actual positions, and each client caches a part of the table on demand. The entry of TT is located by the logical leaf-node number (LLN) and consists of a valid bit, a 31-bit actual leaf-node number (ALN), a 24-bit expected incarnation (INCA), and an 8-bit counter, as shown in Fig. 5. The client can calculate the (host) virtual address of the target leaf node using ALN and the base address of an RDMA-registered memory region. Further, the match of incarnation between TT's entry and target leaf node guarantees that the leaf node has not been reused.

Training models and TT. The server (re-)trains a 2-level ML model (XMODEL) with a translation table (TT) over XTREE's leaf nodes in the background, and each client (re-)fills the learned cache (XCACHE) on demand. Fig. 6 shows

[3] The client may not be the end user but the computation node or the front-end of RDMA-based datacenter applications [34, 35, 16, 17, 25, 55, 57].

[4] Similar to prior RDMA-enabled KVS [16, 52, 35], XSTORE currently allows fixed-length key and fixed/variable-length value. For variable-length value, the leaf node should store a 64-bit fat pointer [16, 53] (the size and the position of value) instead of the value. We discuss how to support variable-length key in §6 and leave it to future work.

```
▸ M: Max. number of sub-models
▸ N: Max. number of keys in each leaf node
XModel {
  Model     top                      ▸ LR: k → [0,1)
  Model[M] subs    ▸ LR: k → [0,pos) w/ min/max_err
}
TRAIN_XMODEL(xmodel)
▸ train top-model
1   cdf = []                           ▸ training set
2   pos = 0
3   foreach k in xtree              ▸ in sorted order
4   | cdf.add(k, pos++)
5   xmodel.top = new LR trained on cdf
▸ assign keys to sub-models
6   kset = [][]       ▸ key set for each sub-model
7   foreach k in xtree
8   | mid = xmodel.top.predict(k) × M
9   | kset[mid].add(k)
▸ train sub-models
10  for i in [0:M)
11  | TRAIN_SUBMODEL(xmodel.subs[i]
    |                MIN(kset[i]), MAX(kset[i]))
TRAIN_SUBMODEL(model, min, max)
12  cdf = []                           ▸ training set
13  LLN = 0                ▸ logic leaf-node number
14  start = xtree.find_lnode(min)
15  end = xtree.find_lnode(max)
16  for lnode in [start:end]
17  | pos = LLN × N
18  | foreach k in lnode.keys   ▸ key-sorted order
19  | | cdf.add(k, pos++)
20  | model.tt[LLN++] = {1, ALN(lnode),
    |                    lnode.inca, lnode.cnt}
21  model = new LR trained on cdf
22  model.calc_err(cdf)     ▸ calculate min/max_err
```

Fig. 6. Pseudo-code of training XMODEL and TT over XTREE.

the pseudo-code of training a complete XMODEL and TT. Starting from a sorted array of keys with logical positions (line 4), we first train the top model. Based on the prediction of the top model, we then evenly partition keys into M sub-models (line 9). Finally, we train each sub-model on a sorted array of its keys with a private logical position at a leaf node granularity (line 12-21) and calculate min- and max-error for every sub-model (line 22). Note that the keys in the leaf node across sub-models will be trained by both of sub-models. Moreover, each sub-model has independent logical positions and an own translation table, making it easy to retrain a sub-model individually when necessary.

In practice, training XMODEL is fast and low-cost, since (1) all of the models in XMODEL are simple linear/multivariate regression models, can be efficiently trained; (2) XMODEL can be partially retrained at a sub-model granularity; and (3) the top model can be trained over a sampled data. As an example, for 100M keys, XMODEL with 500K sub-models takes about 4 seconds to train the top-model and 8 microseconds for each sub-model using a single thread. Further, the client can fill a 500K sub-models XCACHE from scratch in less than one second.

```
LOOKUP(key, &addr)
1   mid = xmodel.top.predict(key) x M
2   model = xmodel.subs[mid]
3   pos = model.predict(key)          ▸ prediction
4   start = (pos - model.min_err)/N     ▸ lnode ID
5   end = (pos + model.max_err)/N       ▸ lnode ID
6   rdma_doorbell = []
7   for n in [start:end]          ▸ from LLN to ALN
8   | entry = model.tt[n]                ▸ TT entry
9   | if entry.valid == 0 then
10  | | return invalid                   ▸ fallback
11  | ra = RA(entry.ALN)           ▸ remote address
12  | rdma_doorbell.add(ra)
    ▸ one RDMA to read disjoint memory regions
13  lnodes = RDMA_READ(rdma_doorbell)
14  for n in [start:end]
15  | lnode = lnodes[n-start]
16  | entry = model.tt[n]
17  | if entry.inca != lnode.inca then
18  | | entry.valid = 0              ▸ invalidation
19  | | return invalid                   ▸ fallback
20  | for i in [0:lnode.cnt)         ▸ local search
21  | | if key == lnode.keys[i] then
22  | | | addr = calc remote addr of ith value
23  | | | return found
24  return not_found              ▸ non-existent key
```

Fig. 7. Pseudo-code of LOOKUP operation based on XCACHE.

A memory-performance trade-off. The ML model is famously memory-efficient. In XMODEL, the basic sub-models are 14B large and consist of two 32-bit floating-point model parameters [5], two 8-bit min- and max-error, and a 32-bit TT size. Thus, XMODEL with 500K sub-models only needs less than 6.7MB. In contrast, TT might dominate the memory usage of XCACHE. For 100M keys, suppose each leaf node has 16 slots (N) and is half-full, TT requires nearly 100MB (15% of the tree-based index). In practice, each client could cache sub-models and TT entries on demand, and even just cache XMODEL to save 99% memory at the cost of 20% performance (using one RDMA READ to fetch a few TT entries).

5.2 Client-direct Operations

In the left part of Fig. 4, XSTORE uses the client-direct design for read requests, namely GET(K) and SCAN(K,N).

5.2.1 GET

Given a key, the client uses XCACHE to lookup the remote position of value using one RDMA READ commonly, replacing RDMA-based traversal in a tree-based index. As shown in Fig. 7, the client first uses XMODEL to predict leaf nodes that cover the lookup key (from `start` to `end`) and then calculates the actual (remote) address of these leaf nodes with TT (line 11). The client can use one RDMA READ with doorbell batching to fetch disjoint memory regions if necessary (line 13).[6] Note that the unit of remote

[5] LR may use more floating-points for prediction.

[6] One RDMA READ can only read a continuous memory region. Yet, we can use an RDMA-aware optimization called doorbell batching [27] to read

read is a leaf node (N keys with a 64-bit header); it is the most likely to read just one leaf node due to the low prediction error of XMODEL. Next, the client uses a local search (e.g., scanning) to find the key from leaf nodes retrieved (line 20-23) and calculates the remote address of the value if it is found (line 22). Finally, the client uses another RDMA READ to fetch the value. Note that any invalid TT entry (line 9 and 17) would result in a fallback path, which ships the GET operation to the server and fetches updated models and TT entries using a single request (i.e., server-centric design).

5.2.2 SCAN

SCAN(K,N) implements a form of range query that returns first (up to) N key-value pairs (in order by key), starting with the next key at or after K. The client first uses the lookup operation with K to determine the remote address of the first key-value pair (larger than or equal to K) and then predicts the leaf nodes that contain the next N key-value pairs, with the help of TT. The translation table provides the number of key-value pairs (CNT) and the actual remote address (ALN) of adjacent leaf nodes (LLN) in sorted order by key. Thus, the client can use one RDMA READ with doorbell batching to fetch these leaf nodes, including keys and values. In general, XSTORE only requires two RDMA READs for each range query. In the rare case, the unexpected result, such as an invalid leaf node (incarnation mismatch) due to dynamic workloads, would cause a fallback path, similar to GET. Note that the range query in XSTORE is also not atomic with respect to updates and inserts as usual [33, 35]; it could be implemented by applications (e.g., transaction [17, 38]).

5.2.3 Non-existent Keys

Intuitively, the ML model guarantees to find all keys have been trained since it stores the worst over- and under-prediction for a CDF (i.e., min- and max-error). However, for non-existent keys, the model should be *monotonic* to guarantee the correct upper and lower bound of a prediction [21, 54], so that a local search could make sure the lookup key does not exist (see line 24 in Fig. 7). Hence, XMODEL adopts monotonic models (e.g., linear regression). As shown in Fig. 8, for a non-existent key (KEY=6), the sub-model LR0 can provide a proper prediction (LR0(6)=[3,4]) that covers the non-existent key (KEYS={5,7}).

However, a hierarchy of models might leave a gap of non-existent keys between neighboring models. Consequently, it still might provide a wrong prediction for these non-existent keys, even if every model is monotonic. For example, the top model selects LR0 for KEY=10 (non-existent), and then LR0 will return a wrong prediction (LR0(10)=[6,7]) that cannot determine whether the key does not exist or the model is out of date from the results (KEYS={17,18}). Worse yet, the non-existent key is common in the range query (e.g., SCAN(K,N)), which demands to retrieve first (up to) N keys larger than or equal to K. As illustrated in Fig. 8, the lookup (LR0(10)) for

multiple disjoint memory regions in one network roundtrip.

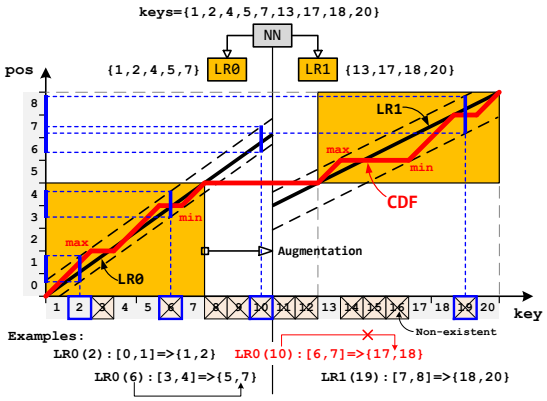


Fig. 8. An example of the prediction for non-existent keys.

a range query SCAN(10,3) will miss a key (KEY=13), so the result (KEYS={17,18,20}) is also wrong.

Data augmentation. To remedy this, we augment the training set of sub-models to cover the gap of non-existent keys between neighboring models. However, data augmentation would increase the prediction error. We thus carefully add a boundary key to both sub-models, which can fill the gaps with minimal overlap between models. For example, in Fig. 8, we add a non-existent key in the gap (KEY=10) with the position of a previous KEY=4 into both sub-models (LR0 and LR1). After that, the lookup of non-existent keys would always return a correct prediction. Further, since the keys in the leaf node across sub-models have been trained by both, there is no need for data augmentation in most cases.

5.3 Server-centric Operations

As shown in the right part of Fig. 4, clients communicate with the server to perform UPDATE(K,V), INSERT(K,V), and DELETE(K) operations; the server updates XTREE concurrently and retrains XMODEL in the background.

Correctness. The correctness condition in XSTORE follows *no lost keys* [33]: the reader must return a correct value for a given key, regardless of concurrent writers. More specifically, when a reader and a writer run concurrently, the reader can return either the old or the new value, while both of them should be atomic.

Concurrency. The hybrid architecture behind XSTORE not only provides separate and appropriate execution paths for static and dynamic workloads (see Fig. 4), but also simplifies the mechanism to guarantee the correctness of concurrent operations. It is critical to the performance of RDMA-based systems due to the lack of richness of RDMA primitives [51]. In Fig. 9a, by using the learned cache (XCACHE), XSTORE restricts (client-direct) remote accesses to the leaf nodes (the dotted red arrow). Thus, we can avoid using sophisticated mechanisms to retrofit a concurrent tree-based index [35].

XTREE reuses an HTM-based concurrent B+tree [50][7] to support concurrent index updates (e.g., node splits) and

[7]The implementation is based on Intel's restricted transactional memory (RTM) that is available as a mature feature in Intel's CPUs (e.g., Skylake).

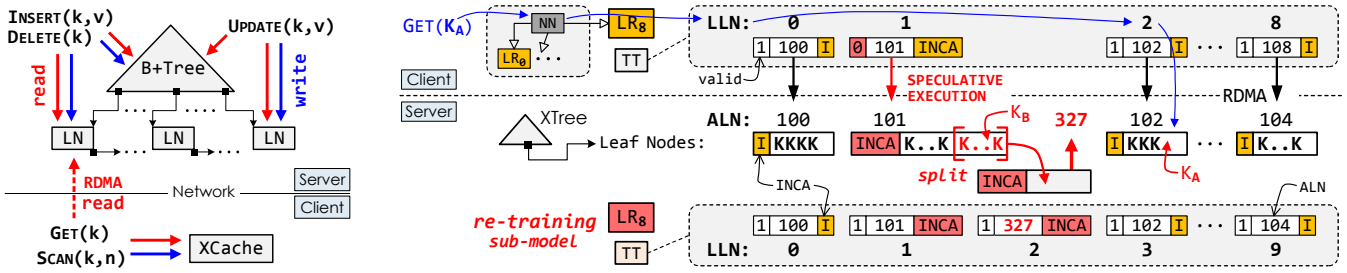


Fig. 9. (a) The access types of different operations for the main components in XSTORE. Red and blue arrows denote read and write accesses. (b) An example of model retraining for LR_8 due to a split of LN_{101}. The leaf node is named by its actual leaf-node number (ALN).

lookups on internal nodes, without the concern of RDMA-based remote accesses. For leaf nodes, XSTORE follows the technique proposed in DrTM+R [11]. Each tree operation at the server is enclosed within an HTM region, that provides strong atomicity in a single machine [5]. In addition, the strong consistency feature of RDMA (where an RDMA operation will abort an HTM transaction that accesses the same memory location [52]) further extends the atomicity when encountering remote accesses. Moreover, as the RDMA operation is only cache-coherent within a cache line, XSTORE adopts versioning [16] for consistent remote reads across multiple cache lines. For the data stored in the leaf node across multiple cache lines, a 16-bit version number is stored both in the header of data and at the start of each cache line. The remote reader matches these versions to detect inconsistent read and must retry if the versions differ. Note that XSTORE hides these versions to applications by automatically converting the data on reads and writes. Finally, the key is also stored in the header of its value, which guarantees consistent remote reads to the key and the value separately.

5.3.1 UPDATE

For UPDATE(K,V), the server first traverses XTREE to the leaf node and updates the value with V if the key (K) exists. Note that the update to the value will not change the index, so that it will also not influence the learned cache and belongs to static workloads.

Optimization: position hint. Although UPDATE(K,V) is a server-side operation, it can still benefit from the learned cache, especially when the server CPU becomes a bottleneck. The client could use XCACHE to predict a position (the remote address of leaf nodes) for the key (see line 1-12 in Fig. 7) and then ship the update request together with the position hint to the server. The server first checks the leaf nodes (by matching incarnation) according to the hint and updates the value if successful. It might skip index traversal and relax the burden on server CPUs. The optimization would increase the performance of update-heavy workloads, like YCSB A (50% update and 50% read).

5.3.2 INSERT and DELETE

INSERT(K,V) and DELETE(K) are shipped to the server and performed on XTREE, as is usual on B+tree. The *in-place* inserts and deletes require moving many key-value pairs within a leaf node to preserve the order of keys. Thus, XTREE chooses not to keep key-value pairs sorted within a leaf node, which can avoid moving key-value pairs and reduces working set in the HTM region. Note that the lookup based on the learned cache will not be affected since it fetches all keys (N) of a leaf node. For DELETE(K), we always overwrite the key and value slot for K with the last key-value pair in the leaf node and update the counter (CNT). Further, the empty leaf node will not be reclaimed to avoid thrashing and model retraining. For INSERT(K,V), we directly append K and V to the key and value slots in the leaf node if K does not exist (see K_A in Fig. 9b). Inserting a key-value pair into a full leaf node will result in a node *split* (see K_B in Fig. 9b). A new leaf node is allocated, and all key-value pairs (plus the new one) are evenly assigned to two leaf nodes in sorted order by key. The original leaf node should increment its incarnation, which makes the clients realize the split. The rest of the split process will execute on the tree index as well as usual.

Retraining and invalidation. The insert of a new leaf node (aka a split) will break the sorted (logical) order of leaf nodes and cause model retraining. An interesting observation behind our solution is that TT decouples model retraining from index updating and allows a *stale* combination of XMODEL and TT to provide a *correct* prediction for the lookup key as long as it is not overlapped with a split. This is because any insert will not cause data movement across leaf nodes, except the split node. For example, LR_8 initially maps K_A to logical node number LN_2, which stores the leaf's physical address 102. After leaf node LN_1 splits due to inserts (a new leaf node with physical address 327), the latest logical node number for K_A is LN_3 after retraining. Yet, the stale TT still maps K_A to physical address 102, the correct position of K_A. Thus, the client can still use a combination of stale models and TTs to find the keys as long as they are not overlapped with split leaf nodes.

Based on this, after a split, the server will *individually* retrain the sub-model and its translation table in the background (see TRAIN_SUBMODEL in Fig. 6) and perform all kinds of operations as usual based on XTREE. Meanwhile, the clients can still directly perform read-only operations based on XCACHE. The incorrect prediction can be detected

by incarnation mismatch between the leaf node and cached TT entry (line 17 in Fig. 7) and results in a fallback, which ships the operation to the server. The client will update XCACHE with a retrained model and its translation table fetched by the fallback. Noted that concurrent splits will not affect model retraining in progress and just make it stale. The new incarnation of the split leaf node ensures the client with this new (stale) model to realize the change of concurrent splits. Each split will issue a retraining task. The training thread currently does not merge or optimize the pending tasks to the same sub-model since it happens very rarely.

Optimization: speculative execution. A split of leaf node just moves the second half of key-value pairs (sorted by key) to its new sibling leaf node. Therefore, the prediction to the split node must still be mapped to this node or its new sibling, like LN_{101} and LN_{327} in Fig. 9b. Based on this observation, *speculative execution* is enabled to handle the lookup operation on a *stale* TT entry (i.e., incarnation check is failed). The client will still find the lookup key in the keys fetched from the split leaf node. If not found, the client will use its right-link pointer to fetch (the second half) keys from its sibling (one more RDMA READ). It means there is roughly half of the chance to avoid incurring a performance penalty. Currently, we only consider one sibling before using a fallback since a cascading split happens rarely. This optimization is important for insert-dominate workloads (e.g., YCSB D) since insert operations and retraining tasks might keep server CPUs busy; the fallbacks will also take server CPU time.

Model expansion. The growing size of key-value pairs in the ML model will likely increase the prediction error, resulting in performance degradation. Prior work [44] uses a sophisticated model split to adapt its learned structure for dynamic workloads, which demands physical data moving and atomic top-model replacement. In response to this problem, XSTORE supports *model expansion* that increases the number of sub-models in XMODEL at once (e.g., doubling) when necessary (e.g., exceeding a threshold of min- and max-error). The model expansion requires a complete training (see Fig. 6) on XTREE to build a new version of XMODEL and TT. Note that model expansion will not affect any requests performed by both the server and the client for several reasons. First, training models will not change or move data. Second, the top model can be trained over incomplete data. Third, the conflicting sub-model retraining could be made up later. Finally, the client can use the originally learned cache during model expansion. Moreover, after deleting a large number of key-value pairs, XSTORE can also resize XMODEL to shrink the number of sub-models using a similar process.

5.4 Durability

XSTORE should log writes (updates, inserts, and deletes) to log files stored in reliable storage for persistence and failure recovery (e.g., server's local disk). As RDMA-based remote accesses are restricted to reads (lookups, gets, and scans), they will not involve in logging and recovery. In addition, XMODEL and TT are tightly associated with XTREE (e.g., virtual address). Thus, they should be rebuilt after recovery.

To ensure correct recovery from a machine failure, XSTORE can reuse the existing durability mechanism in the concurrent tree-based index extended by XTREE, like version numbers [50, 33]. Each worker thread at the server appends the log (key, value, and version) to its in-memory log buffer. A corresponding logging thread, sharing the same core with the worker thread, writes out the log buffer to its log file in the background. The logger batches the log entries to avoid the storage backend becoming the bottleneck. During recovery, XSTORE scans log files to sort logs of the same key by its version number and applies the latest log of keys in parallel. Finally, XSTORE rebuilds XMODEL and TT by training over recovered XTREE.

5.5 Scaling out XSTORE

XSTORE follows a coarse-grained scheme [57], the dominant solution, to distribute an ordered key-value store span multiple servers (scale-out). XSTORE first assigns key-value pairs to the servers based on a range-based partitioning function for the keys. Then each server constructs XTREE individually for its assigned key-value pairs and further trains a corresponding XMODEL and TT. Note that the boundary keys should be added to the training set to cover the gap of non-existent keys between neighboring servers.

The client maintains a separate learned cache for each server and uses the same partitioning function to decide which server should perform a given request. Based on it, the client can perform requests as mentioned in §5.2 and §5.3, with one exception—SCAN(K,N) reads a range of key-value pairs span multiple servers. After the lookup of K on a specified server, the client might find that the expected number (N) exceeds the remaining key-value pairs in this server. Starting from the first logical leaf node on the next server, the client can predict the leaf nodes that contain the rest of key-value pairs. Finally, the client uses one RDMA READ for each server involved to fetch these leaf nodes.

6 Discussion

Support variable-length keys. XSTORE currently supports fixed-length key and variable/fixed-length value. To support variable-length key, XSTORE should store a fat pointer in the leaf node of XTREE (instead of the actual key), which encodes the size and position of the key. This scheme can traverse variable-length key locally by CPUs (i.e., server-centric design), while it would be hard to do it efficiently by using one-sided RDMA READs (i.e., client-direct design). XSTORE has to retrieve the actual keys using an additional RDMA READ for each (Line 21 in Fig. 7). Therefore, XSTORE further stores a fixed hash code of the key within the fat pointer. Consequently, the client could directly compare the hash codes instead of keys, after fetching the leaf node for a given key. Note that the actual (variable-length) key should

Table 1: YCSB workload description. **R**, **U**, **I**, **M**, and **S** denote read, update, insert, read-modify-update, and scan, respectively. Scan accesses N values, where N is uniformly distributed in [1,100].

YCSB	A	B	C	D	E	F
Type	R : U	R : U	R	R : I	S : I	R : M
Ratio (%)	50 : 50	90 : 10	100	95 : 5	95 : 5	50 : 50

be checked to avoid a hash collision. For example, the client can fetch the value associated with the key. We plan to extend XSTORE to support variable-length keys in future work.

Data distribution. XSTORE assumes machine learning (ML) models can effectively learn various data distributions (e.g., log-normal [29, 44, 15]). Based on it, we believe there is a trade-off among the memory consumption of XCACHE, the retraining costs of XMODEL, and the performance of XSTORE. When using simple models (e.g., linear regression) for fast model retraining, XSTORE has to use many models to achieve high accuracy for irregular data distributions. For such a scenario, clients can only cache partial sub-models due to the increased model memory consumptions. On the other hand, XSTORE could use complex models (e.g., neural network (NN)) to achieve high accuracy with few models. Yet, NN is slow on model retraining and may impact the performance under dynamic workloads (e.g., inserts), since the client may fall back more often due to stale XCACHE.

7 Evaluation

7.1 Experimental Setup

Testbed. Without explicit mention, we use one server machine and (up to) 15 client machines. Each machine has two 12-core Intel Xeon CPUs, 128GB of RAM, and two ConnectX-4 100Gbps IB RNICs. Each RNIC is used by threads on the same socket and connected to a Mellanox 100Gbps IB Switch. The server registers the memory with huge pages to reduce RNIC's page translation cache misses [16].

Workloads. We use YCSB [13] and two production workloads from Nutanix [30]. We mainly focus on YCSB as it contains various types of workloads [12]: update heavy (A), read mostly (B), read only (C), read latest (D), short ranges (E), and read-modify-write (F). Table 1 shows a summary of YCSB workloads (A-F). Since small requests dominate in real-life workloads [4], we evaluate KV stores with 100 million KV pairs initially (a 7-level tree-based index and a leaf level), where 8-byte key and 8-byte value are used, similar to prior work [33, 35, 24, 44]. Both Uniform and Zipfian key distributions are evaluated for all YCSB workloads. Note that YCSB D only has Uniform and Latest key distributions; the client is likely to query its recently inserted keys in Latest distribution. In addition, each client generates their insert key uniformly and randomly in YCSB D and E. The two production workloads both have a profile of 57:41:2 write:read:scan ratio, while the access patterns of them are relatively uniform (Prod1) and skewed (Prod2), respectively. Both of them have 500 million KV pairs with 8-byte key and 64-byte value. Finally, besides the default data distribution of the above workloads, we also use two synthetic and one real-life datasets (see Table 2) to study the behavior of learned cache in depth.

Table 2: Data distribution description for evaluating datasets.

Name	Description	Workloads
L	Linear	YCSB[13], Nutanix[30]
NL	Noised linear	YCSB[13]
OSM	Longitude location	Open Street Map[2]

Comparing targets. We compare XSTORE to three state-of-the-art RDMA-based ordered KV stores: DrTM-Tree [11] and eRPC+Masstree [24] (*server-centric* design), as well as Cell [35] (*client-direct* design). eRPC+Masstree (EMT) adopts eRPC [24] (RDMA-based RPC library) to extend Masstree [33] (in-memory ordered KV store). We implement DrTM-Tree and Cell in the same framework to provide an apple-to-apple comparison with two typical designs, but also because DrTM-Tree uses similar B+tree [50] and RDMA library [51] with XSTORE, and Cell is not open-source.[8] We further consider RDMA-Memcached v0.9.6 [22] (RMC) in our experiments, which is an RDMA version of memcached [1], a widely used network-attached KV in industry.

All systems fully utilize all of the 24 CPU cores (with hyperthreading disabled) and two RNICs. As EMT and RMC cannot use multiple NICs simultaneously, we deploy two instances at the server on different sockets, and each instance uses the RNIC attached to that socket. This actually makes them faster during experiments since it avoids cross-socket synchronizations. XSTORE uses (up to) two auxiliary threads to train ML models in the background for dynamic workloads. XTREE is configured with a fanout of 16. XMODEL uses 500K sub-models for static workloads and 2M models for dynamic workloads to avoid model expansion during evaluation (because XSTORE can insert more than 150M KV pairs in 60s). In addition, logging is disabled in all systems, and the server hosts all data in main memory.

7.2 YCSB Performance

Fig. 10 compares the peak throughput of various RDMA-based key-value stores for YCSB with Uniform and Zipfian distributions, where all systems are saturated by up to 15 client machines. Note that RMC performs poorly in all experiments as it is bottlenecked by CPU synchronizations [43, 31]. Due to space limitations, we skip detailed discussion of experimental results on it.

Read-only workload (YCSB C). For Uniform distribution, XSTORE can achieve 82 million requests per second, even a little higher than the *optimal* throughput (a whole-index

[8] For DrTM-Tree, our experimental results were confirmed by the authors. For Cell, we follow the same caching strategy—the client caches nodes at least four levels above the leaf node at the clients with LRU policy to minimize churn and maximize hits. Based on a comparison against published numbers, we believe that the large performance difference between XSTORE and other systems (e.g., 27M reqs/s from our implementation vs. 0.95M reqs/s from Cell [35] for YCSB A with Zipfian distribution) offsets performance variations due to system and implementation details.

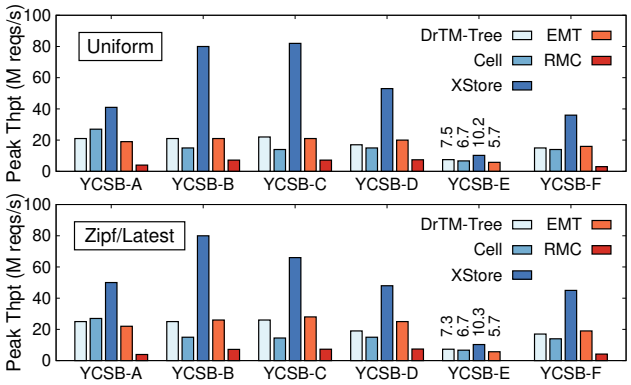


Fig. 10. Comparison of throughput on various RDMA-based KVs using YCSB. Note that RMC does not support range queries.

cache), since it only uses one RDMA READ to fetch one leaf node per lookup; the payload is 16B smaller by avoiding a sophisticated mechanism for consistency (i.e., min-max fence keys [35]). The prediction error of XCACHE is just 0.74. This number outperforms EMT, DrTM-Tree, and Cell by 3.9×, 3.7×, and 5.9×, respectively. Both DrTM-Tree and EMT are bottlenecked by server CPUs, while Cell is bottlenecked by RDMA amplifications; it still needs four RDMA READs to traverse tree nodes even index caching is enabled.

For Zipfian distribution, XSTORE can still outperform EMT, DrTM-Tree, and Cell by 2.4×, 2.5×, and 4.6×, respectively. The systems with server-centric design perform better due to better CPU cache locality. However, the peak throughput of XSTORE drops by 18% since RDMA has relatively poor performance when massive clients read a small range of memory simultaneously. We suspect that our current RNIC (ConnectX-4) checks conflicts between one-sided RDMA operations based on request's address [27], so that these operations may compete for NIC's internal processing resources, even if there is no conflict.

Static read-write workloads (YCSB A, B, and F). For update-heavy workloads (YCSB A), XSTORE is still bottlenecked by server CPUs for handling updates. However, compared to server-centric KVs (e.g., DrTM-Tree and EMT), the clients in XSTORE can directly perform read requests with the help of learned cache, which completely bypasses server CPUs. Therefore, XSTORE can still provide up to 2.2× and 2.3× (from 1.5× and 2.0×) throughput improvements for Uniform and Zipfian distributions, respectively, compared to other KVs. For read-mostly workloads (YCSB B), the speedup of throughput in XSTORE further reaches up to 5.3× (from 3.1×). There are two reasons: (1) the read requests are less skewed interleaved with (10%) updates, compared to read-only workloads (YCSB C); (2) the server of XSTORE has not been saturated (less than 40% of CPU utilizations); thus it is still sufficient to perform updates, compared to update-heavy workloads (YCSB A). The performance of XSTORE on YCSB F is somewhere in between since it has about 75% reads.

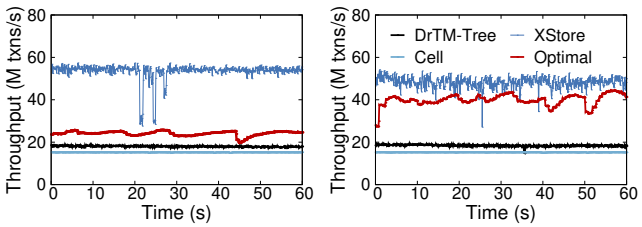


Fig. 11. The performance timeline of YCSB D with (a) Uniform and (b) Latest workloads.

Dynamic workloads (YCSB D and E). The throughput of every system is impacted by dynamic workloads due to the contention between reads and inserts. For DrTM-Tree and EMT, the contention happens on the tree-based index. For XSTORE and Cell, the performance slowdown is mainly due to cache invalidations. However, Cell only caches the top four levels, where node split is rare. The overhead in XSTORE mainly comes from two parts: (1) cache invalidations would increase RDMA operations due to fallbacks (RDMA-based RPC) and speculative execution (50% one more RDMA READ); (2) a dynamic dataset is always harder to learn than a static dataset due to the randomly inserted new keys; the prediction error would stably increase to 8.3 for YCSB D.[9] Fortunately, the clients can still use stale learned cache for most read requests, and model retraining is also very fast. Thus, for YCSB D, XSTORE can provide up to 3.5× and 3.2× (from 2.7× and 1.9×) speedup and achieve 53M and 48M reqs/s throughput for Uniform and Latest distributions, respectively. For YCSB E, the performance is dominated by scanning a large range of KV pairs. Thereby the difference is relatively small, and XSTORE outperforms other systems by up to 1.8× (from 1.4×).

Fig. 11 further shows the timelines for YCSB D with Uniform and Latest workloads. The optimal throughput of tree-based index cache can only achieve about 25M reqs/s, more than 3× lower than its read-only throughput (78M reqs/s), and suffers from severe performance fluctuations due to frequent cache invalidations, especially for Uniform distribution. For Latest distribution, each client will focus on a small range of KV pairs (latest inserted by itself), which significantly reduces cache misses and invalidations due to accessing internal nodes split by other clients. XSTORE preserves relatively high throughput and has steady cache invalidation rates, 5% for Uniform, and 21% for Latest. It is mainly because stale learned cache can still provide a correct prediction for most read requests. The speculative execution also helps to halve the rate (from 10% to 5%). In addition, in Latest distribution, each client will frequently access KV pairs just inserted. If the insert incurs a node split, XSTORE might not fetch a new model immediately (wait for model retraining) and would increase cache misses.

***CPU utilizations of* XSTORE.** Note that XSTORE uses two auxiliary threads to retrain XMODEL for dynamic work-

[9] The data distribution of dynamic workloads (i.e., YCSB D and E) is close to noised linear (NL). Hence, XSTORE can only achieve 61M reqs/s for YCSB D with 2M models even no inserts (see Fig. 14b and Fig. 15d).

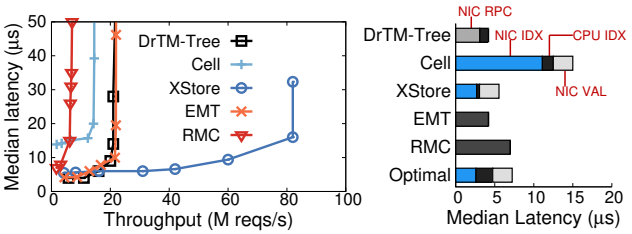


Fig. 12. Comparison of (a) throughput-latency and (b) end-to-end median latency at low load for YCSB-C with a uniform distribution.

loads, causing increased server CPU usage. Yet, XSTORE still saves server CPUs compared to server-centric KVs (e.g., DrTM-Tree) due to handling read requests in the clients. For example, DrTM-Tree saturates all CPUs (24 × 100%) for YCSB D, while XSTORE just consumes under half for serving insert requests and retraining sub-models.

End-to-end latency. Fig. 12a shows the throughput-latency curves for YCSB C with a uniform distribution. Due to space limitations, we omit other workloads that are similar. When using few clients (low load), server-centric KVs have lower latency, as one RPC round trip is faster than two one-sided RDMA operations, namely DrTM-Tree (NIC_RPC) vs. XSTORE (NIC_IDX and NIC_VAL) in Fig. 12b. However, the throughput of them (e.g., DrTM-Tree) is saturated by CPUs much earlier (about 20M reqs/s), and the latency would rapidly collapse. On the other hand, the latency of Cell is limited by multiple RDMA READs for each lookup (NIC_IDX) even at low load. In contrast, XSTORE only needs one RDMA READ, thanks to the learned cache. As a reference, we provide the latency of using whole-index cache (Optimal) that also takes just one RDMA READ. However, traversing tree-based index locally still takes more time (2.14µs in CPU_IDX) due to many random memory accesses, compared to XSTORE (0.35µs). Moreover, XSTORE can keep low latency at much high load (82M reqs/s with median latency of 16µs) by eliminating CPU bottleneck at the server.

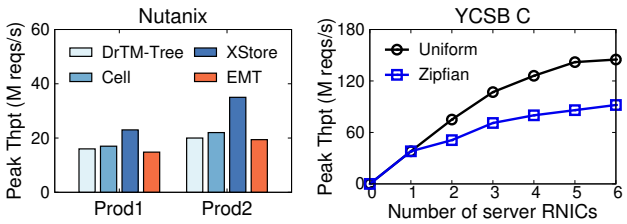


Fig. 13. (a) Performance comparison with production workloads. (b) Scalability of XSTORE on YCSB C with the increase of RNICs.

7.3 Production Workload Performance

Fig. 13a shows the peak throughput of XSTORE and other systems on two write-intensive production workloads, similar to YCSB A. The performance is also mainly bottlenecked by server CPUs due to 57% of writes. In the first workload (Prod1), XSTORE outperforms DrTM-Tree, EMT, and Cell by 1.44×, 1.55×, and 1.35×, respectively. The speedup in the second workload (Prod2) increases to 1.75×, 1.80×, and 1.60× since this workload is more skewed.

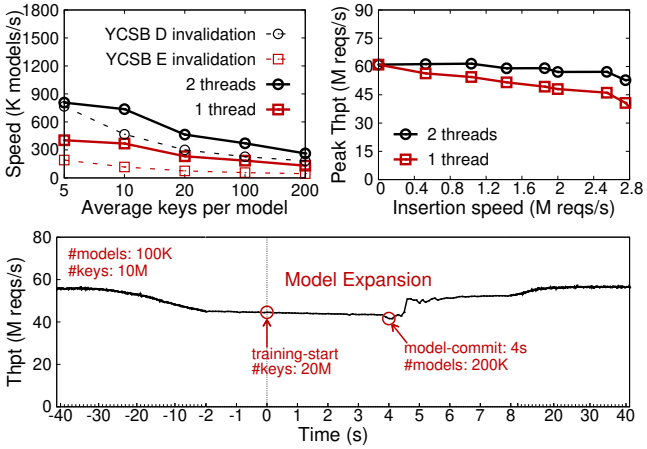


Fig. 14. (a) Comparison between sub-model retraining and invalidation speed. (b) Performance of XSTORE with the increase of insertion speed. (c) Performance timeline with model expansion.

7.4 Scale-out Performance

Fig. 13b shows the scalability of XSTORE with up to 6 server RNICs (3 server machines). We scale XSTORE by range-based partitioning a YCSB dataset with 600M keys into different numbers of RNICs. The performance is measured using up to 13 client machines (26 RNICs) with a read-only workload. For a uniform request distribution, XSTORE achieve a peak throughput of 145M reqs/s, which is limited by the number of client machines. Note that, on our testbed, XSTORE needs about eight client RNICs to saturate one server RNIC. XSTORE scales to 1.97× and 2.81× by using 2 and 3 server RNICs, respectively. For a skewed request distribution (Zipfian), XSTORE just reaches 92M reqs/s by using 6 server RNICs since most requests (more than 35%) are sent to one RNIC. It throttles the entire system.

7.5 Model (Re-)Training and Expansion

Fig. 14a shows the throughput of training models using one or two threads and model invalidation speed for dynamic workloads (YCSB D and E). Empirically, using two threads for model retraining is sufficient for XSTORE to reach a throughput of 53M reqs/s (YCSB D). XSTORE can retrain sub-models individually and takes 8µs on average to retrain a model with 200 keys. Note that the insertion speed reaches about 2.65M reqs/s for YCSB D (5% inserts). For dynamic workloads, the throughput of XSTORE would decrease when stale sub-models can not retrained in time. To quantify the performance overhead, we evaluate XSTORE with the increase of insertion speed, similar to YCSB D (except that one client is dedicated to insert key-value pairs with a given speed, and the rest of clients still issue reads). As shown in Fig. 14b, the throughout drops below 40% (61M vs. 37M reqs/s) under the peak insertion speed (2.8M reqs/s, limited by server CPUs) when using a single retraining thread. Further, when using two threads, the performance degradation is limited to 13%.

Finally, the growing size of KV pairs in the ML model

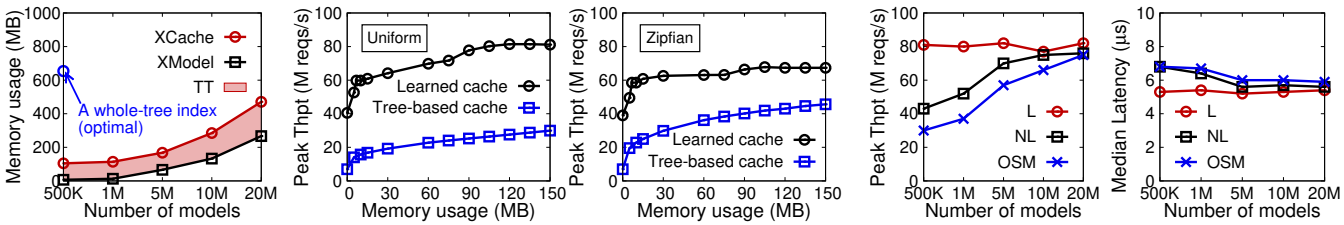


Fig. 15. (a) Memory usage of learned cache (XCACHE). Comparison of peak throughput between learned cache and tree-based cache with different memory footprint at the client for YCSB C using (b) Uniform and (c) Zipfian distributions. Comparison of (d) peak throughput and (e) median latency on XSTORE with the increase of models for various data distributions (see Table 2).

will likely increase the prediction error, resulting in performance degradation. XSTORE supports model expansion to increase models in the background if needed. As shown in Fig. 14c, starting from 10M keys and 100K models, several clients continuously insert KV pairs, and the performance of XSTORE slowly degrades for read requests. When the average number of keys per model exceeds 200 (a user-defined threshold), the server starts to train a new XMODEL with double sub-models (200K) in the background from 0s to 4s, with negligible overhead. After that, the server will commit the new model, and clients could individually fetch new sub-models on demand. The performance resumes rapidly in 2s.

7.6 Memory Footprint of XCACHE

Fig. 15a presents the memory usage of XCACHE with the increase of sub-models for 100M KV pairs. Note that the entire XTREE has 654MB internal nodes. The size of TT depends on the number of leaf nodes. Since each leaf node has 16 slots for KV pairs, TT occupies around 98MB as the tree-based index is half-full. Thus, TT would dominate the memory usage for a small XMODEL since each sub-model is 14B large. To achieve peak throughput, XMODEL with 500K sub-models is enough for read-only workloads (YCSB C) with 100M KV pairs, while it needs 2M sub-models for dynamic workloads (YCSB D) with 250M KV pairs.

As shown in Fig. 15b and Fig. 15c, compared to conventional tree-based index cache, XSTORE can provide competitive performance with much lower memory footprint at the clients, even (almost) no memory footprint. XCACHE prefers to store XMODEL, which may only occupy 1% memory (6.8M vs. 654MB). It means that, for YCSB C with Uniform and Zipfian distributions, XSTORE can achieve 74% and 87% of optimal throughput (a whole-index cache), where the client uses one additional RDMA READ to fetch several 8-byte TT entries for each lookup. Even if the client only stores a 16-byte top model, XSTORE can still achieve about 40M reqs/s by using one RDMA READ to fetch a 14-byte sub-model first.

7.7 Data Distribution

We further evaluate XSTORE on a 100M-key dataset with different data distributions in Table 2 using a read-only workload (YCSB C). The throughput of XSTORE is sensitive to the prediction error due to bandwidth amplification for retrieving more keys. Thus, XSTORE requires more simple sub-models (e.g., LR) to learn complex data distributions (e.g., OSM) for the same prediction error. For example, as shown in Fig. 15d, XSTORE requires about 20M sub-models for OSM to achieve a peak throughput of 80M reqs/s. However, as shown in Fig. 15e, the median latency at a low load is relatively stable for various data distributions, as the latency of RDMA is insensitive to payload sizes when the network is not saturated [39].

Table 3: The impact of durability on throughput (M reqs/s).

YCSB /Uniform	A	B	C	D	E	F
w/o logging	41	80	82	53	10.2	36
w logging	31	78	82	51	9.9	33

7.8 Durability

To study the overhead of logging for durability, we evaluate the peak throughput of XSTORE for various YCSB workloads with logging to SSD enabled. As shown in Table 3, the performance drops by up to 24% for update-heavy workloads (e.g., YCSB A) due to additional writes to SSD for write operations (e.g., UPDATE). On the other hand, it does not degrade the performance of read-heavy workloads much (e.g., YCSB C). First, XSTORE executes read operations (e.g., GET) using one-sided RDMA primitives, bypassing the logging threads thoroughly. Second, XSTORE flushes the logs in a batched manner [33], which hides the impact of slow storage (§5.4).

7.9 Variable-length Value

By default, XSTORE directly stores the value in leaf nodes (*inline value*). To support variable-length values, XSTORE stores a 64-bit fat pointer (the size and the position of value) in leaf nodes (*indirect value*). Consequently, the client needs an additional RDMA READ to retrieve the variable-length value (Line 13 in Fig. 7). Fig. 16a shows the performance of XSTORE by using inline and indirect value. Using *indirect value* causes up to 43% (from 8%) performance degradation, compared to using *inline value*. The performance gap is closing with the increase of values (e.g., 1KB) since the cost of one additional RDMA READ becomes trivial.

7.10 Application Performance

To demonstrate the effectiveness of XSTORE in application workloads, we have integrated it into DrTM+H [51], a state-

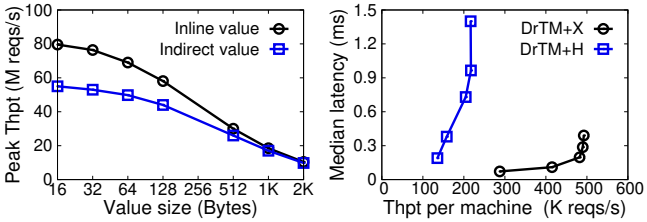


Fig. 16. (a) Performance of XSTORE by using inline and indirect value. (b) Comparison of DrTM+H on TPC-C w/ and w/o XSTORE.

of-the-art distributed OLTP system that leverages RDMA-enabled KVS to store tuples. The vanilla DrTM+H only performs unordered index lookups (hash table) by using one-sided RDMA primitives [52]. DrTM+H with XSTORE (called DrTM+X) can further perform ordered index lookups (B+tree) through one-sided RDMA operations.

Experimental setup. We use TPC-C [45] to compare the performance of DrTM+H and DrTM+X. Note that both of them run in an asymmetric setting, which is widely adopted in cloud databases [55, 47, 7].[10] More specifically, we deploy 96 warehouses on four data servers and use the rest of the machines in our testbed as clients. Both DrTM+H and DrTM+X rely on the data server to update tuples, while DrTM+X uses one-sided RDMA READs to retrieve tuples from the data server. Therefore, we use a read-heavy TPC-C workload in the experiment, which consists of NEW-ORDER transactions (10%) and ORDER-STATUS transactions (90%). NEWORDER transaction inserts a new order with five to fifteen order lines; ORDERSTATUS transaction retrieves the recently inserted orders first and then scans related order lines.

Performance. As shown in Fig. 16b, XSTORE improves the peak throughput of DrTM+H by 2.27×, reaching 490K reqs/s. DrTM+H is bottlenecked by server CPUs since the data server traverses the index and performs the read request locally. Consequently, the read requests of ORDERSTATUS transactions would compete CPUs with the write requests of NEWORDER transactions at the servers. Differently, DrTM+X relies on RNICs at the clients to lookup and retrieve tuples for ORDERSTATUS transactions. It relaxes the burden on server CPUs and improves performance significantly.

8 Related Work

RDMA-enabled key-value stores. XSTORE continues the line of research of RDMA-based in-memory key-value stores [31, 34, 25, 16, 52, 35, 43, 57, 8, 48], but explores a new design point, namely *learned cache*, that leverages machine learning (ML) models as index cache for RDMA-based, tree-backed key-value store. There have been many efforts to investigate RDMA-based unordered in-memory KVs which focus on such as improving the communication layer (e.g.,RPC) [25, 24, 10], selecting appropriate hash tables [34, 16, 52], supporting index caching [52, 48], and enabling in-network processing [31, 40].

[10]Prior work [51] has shown that using (two-sided) RDMA-based RPC is a better choice for GET operations in a symmetric setting [17].

There is an increasing interest in optimizing tree-backed in-memory key-value stores with RDMA. Cell [35] allows clients to traverse server's B+Tree using RDMA READs and caches the top three levels of tree index. FaRM B-Tree [17] caches B-tree's internal nodes at each server to accelerate lookups using RDMA, while it is costly and error-prone for dynamic workloads [38]. Ziegler et al. [57] studies different RDMA-based design alternatives for tree-based index, including how the tree should be distributed and the choices of RDMA primitives for tree operations.

Learned indexes and their applications in systems. Kraska et al. [29] argue that all existing index structures can be replaced with machine learning (ML) models, which are termed "learned index", and further propose several example learned indexes to replace various index structures, including tree-based range index. There have been several recent efforts of adapting learned indexes to handle dynamic workloads [44, 15, 36]. XIndex [44] adds a delta index to each sub-model in a learned index and proposes a new concurrent compaction scheme to split models. ALEX [15] uses a gapped array to accommodate new key-value pairs, similar to the leaf node of XTREE. However, it is non-trivial to enable the gapped array in a distributed system since it requires complex coordinations when expanding the array upon full. Bourbon [14] is a log-structured merge (LSM) tree that leverages the learned index to speedup lookups. FITING-TREE [18] is a form of a learned index to balance prediction error and memory cost. It uses extra sorted buffers to store inserts and merges them back when reaching a threshold. SIndex [49] is a concurrent learned index for variable-length string keys. Differently, XSTORE proposes a hybrid architecture to leverage ML models as RDMA-based index cache, instead of replacing or augmenting traditional index structures.

9 Conclusion

This paper presents XSTORE, an RDMA-based in-memory ordered key-value store with a new *hybrid* architecture to leverage ML model as RDMA-based index cache. Our experimental results show the high performance of XSTORE.

10 Acknowledgment

We sincerely thank our shepherd Andrea C. Arpaci-Dusseau and the anonymous reviewers for their insightful suggestions. We also thank Zhaoguo Wang, Chuzhe Tang, Zhiyuan Dong and Youyun Wang for sharing their experience on *learned* index, and Xiating Xie for the valuable feedback. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003), the National Natural Science Foundation of China (No. 61772335, 61925206, 61732010), the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and a research grant from Huawei Technologies. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] Memcached. https://memcached.org/.

[2] OpenStreetMap (OSM) on AWS. https://aws.amazon.com/public-datasets/osm, 2020.

[3] AGUILERA, M. K., KEETON, K., NOVAKOVIC, S., AND SINGHAL, S. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2019), HotOS '19, Association for Computing Machinery, p. 120–126.

[4] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[5] BLUNDELL, C., LEWIS, E. C., AND MARTIN, M. M. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters 5*, 2 (2006).

[6] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H. C., ET AL. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference* (2013), pp. 49–60.

[7] CAO, W., LIU, Z., WANG, P., CHEN, S., ZHU, C., ZHENG, S., WANG, Y., AND MA, G. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment 11*, 12, 1849–1862.

[8] CASSELL, B., SZEPESI, T., WONG, B., BRECHT, T., MA, J., AND LIU, X. Nessie: A decoupled, client-driven key-value store using rdma. *IEEE Transactions on Parallel and Distributed Systems 28*, 12 (2017), 3537–3552.

[9] CHEN, H., CHEN, R., WEI, X., SHI, J., CHEN, Y., WANG, Z., ZANG, B., AND GUAN, H. Fast in-memory transaction processing using rdma and htm. *ACM Trans. Comput. Syst. 35*, 1 (July 2017).

[10] CHEN, Y., LU, Y., AND SHU, J. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[11] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 26.

[12] COOPER, B. F. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[13] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC'10, ACM, pp. 143–154.

[14] DAI, Y., XU, Y., GANESAN, A., ALAGAPPAN, R., KROTH, B., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. From wisckey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation* (2020), OSDI '20, USENIX Association.

[15] DING, J., MINHAS, U. F., YU, J., WANG, C., DO, J., LI, Y., ZHANG, H., CHANDRAMOULI, B., GEHRKE, J., KOSSMANN, D., LOMET, D., AND KRASKA, T. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 969–984.

[16] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14, USENIX Association, pp. 401–414.

[17] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP'15, ACM, pp. 54–70.

[18] GALAKATOS, A., MARKOVITCH, M., BINNIG, C., FONSECA, R., AND KRASKA, T. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1189–1206.

[19] GRAEFE, G. Write-optimized b-trees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases* (2004), VLDB '04, VLDB Endowment, p. 672–683.

[20] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM'16, ACM, pp. 202–215.

[21] GUPTA, M., COTTER, A., PFEIFER, J., VOEVODSKI, K., CANINI, K., MANGYLOV, A., MOCZYDLOWSKI, W., AND VAN ESBROECK, A. Monotonic calibrated interpolated look-up tables. *J. Mach. Learn. Res. 17*, 1 (Jan. 2016), 3790–3836.

[22] HIGH-PERFORMANCE BIG DATA (HIBD). RDMA-based Memcached (RDMA-Memcached). http://hibd.cse.ohio-state.edu.

[23] JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C.-C., KHANDELWAL, A., PU, Q., SHANKAR, V., CARREIRA, J., KRAUTH, K., YADWADKAR, N., ET AL. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[24] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter rpcs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 1–16.

[25] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM'14, ACM, pp. 295–306.

[26] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association, pp. 185–201.

[27] KAMINSKY, A. K. M., AND ANDERSEN, D. G. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference* (2016), p. 437.

[28] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (USA, 2018), OSDI'18, USENIX Association, p. 427–444.

[29] KRASKA, T., BEUTEL, A., CHI, E. H., DEAN, J., AND POLYZOTIS, N. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data* (2018), ACM, pp. 489–504.

[30] LEPERS, B., BALMAU, O., GUPTA, K., AND ZWAENEPOEL, W. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 447–461.

[31] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 137–152.

[32] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 429–444.

[33] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys'12, ACM, pp. 183–196.

[34] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[35] MITCHELL, C., MONTGOMERY, K., NELSON, L., SEN, S., AND LI, J. Balancing cpu and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[36] NATHAN, V., DING, J., ALIZADEH, M., AND KRASKA, T. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 985–1000.

[37] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., ET AL. Scaling memcache at facebook. In *nsdi* (2013), vol. 13, pp. 385–398.

[38] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIĆ, A., NARAYANAN, D., AND CASTRO, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 433–448.

[39] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 317–332.

[40] SIDLER, D., WANG, Z., CHIOSA, M., KULKARNI, A., AND ALONSO, G. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[41] SOWELL, B., GOLAB, W., AND SHAH, M. A. Minuet: A scalable distributed multiversion b-tree. *Proc. VLDB Endow. 5*, 9 (May 2012), 884–895.

[42] SREEKANTI, V., WU, C., LIN, X. C., SCHLEIER-SMITH, J., GONZALEZ, J. E., HELLERSTEIN, J. M., AND TUMANOV, A. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow. 13*, 12 (jul 2020), 2438–2452.

[43] SU, M., ZHANG, M., CHEN, K., GUO, Z., AND WU, Y. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 1–15.

[44] TANG, C., WANG, Y., DONG, Z., HU, G., WANG, Z., WANG, M., AND CHEN, H. Xindex: A scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2020), PPoPP '20, Association for Computing Machinery, p. 308–320.

[45] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[46] TSAI, S.-Y., AND ZHANG, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.

[47] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 1041–1052.

[48] WANG, Y., MENG, X., ZHANG, L., AND TAN, J. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SoCC'14, ACM, pp. 23:1–23:13.

[49] WANG, Y., TANG, C., WANG, Z., AND CHEN, H. Sindex: A scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2020), APSys '20, Association for Computing Machinery, p. 17–24.

[50] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys'14, ACM, pp. 26:1–26:15.

[51] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI '18, pp. 233–251.

[52] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[53] XIE, X., WEI, X., CHEN, R., AND CHEN, H. Pragh: Locality-preserving graph traversal with split live migration. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 723–738.

[54] YOU, S., DING, D., CANINI, K., PFEIFER, J., AND GUPTA, M. Deep lattice networks and partial monotonic functions. In *Advances in neural information processing systems* (2017), pp. 2981–2989.

[55] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow. 10*, 6 (Feb. 2017), 685–696.

[56] ZHANG, H., ANDERSEN, D. G., PAVLO, A., KAMINSKY, M., MA, L., AND SHEN, R. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data* (2016), ACM, pp. 1567–1581.

[57] ZIEGLER, T., TUMKUR VANI, S., BINNIG, C., FONSECA, R., AND KRASKA, T. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, ACM, pp. 741–758.

A Artifact Appendix

A.1 Abstract

This artifact provides the source code of XSTORE and scripts to reproduce the main experimental results. XSTORE is an RDMA-based ordered key-value store that adopts the client-server model (network-attached) and range index structures (tree-backed). To reproduce the results, we provide instructions to build binaries (§A.3) and run experiments (§A.4). The source code of XSTORE can be retrieved from a public open-source repository (§A.2.1). The repository also contains scripts to generate the main results in §7 (see Table 4). Though the scripts target our testbed (§7.1), readers can simply change them for other platforms (§A.6).

A.2 Artifact Check-list

- **Program:** `fserver`, `ycsb`, and `micro`.
- **Compilation:** `g++` and `cmake`.
- **Hardware:** Intel CPU with RTM and Mellanox NIC with RDMA.
- **Execution:** Python scripts.
- **Metrics:** Throughput and median latency.
- **Expected experiment run time:** 1 minute each experiment.
- **Public link:** https://github.com/SJTU-IPADS/xstore.
- **Code licenses:** Apache License 2.0.

A.2.1 How to Access

The artifact is publicly available at our Github repository.

```
$ git clone https://github.com/SJTU-IPADS/xstore
$ git checkout c9f38188
```

A.2.2 Hardware Dependencies

To reproduce the experiment results, each machine must have at least one Mellanox RDMA network card (e.g., Mellanox ConnectX-4 MT27700 100Gbps InfiniBand NIC), and the server machine must have Intel processors with Restricted Transactional Memory (RTM) (e.g., Xeon E5-2650 v4). It should be noted that the throughput of read operations (e.g., gets) is mainly bottlenecked by the RDMA network, while the throughput of write operations (e.g., updates) is mainly bottlenecked by the server CPU.

A.2.3 Software Dependencies

Operating system: `Ubuntu` $\geq$ `16.04`.

Compile toolchain: `g++` $\geq$ `5.4.4` and `cmake` $\geq$ `3.5.1`.

Other software dependencies: `Intel MKL`, `Mellanox OFED`, `boost 1.6.1`, and `jemalloc`.

A.3 Installation

Intel MKL (Math Kernel Library).

```
$ apt-get install -y intel-mkl-2019.1-053
```

Listing 1: A sample evaluating script (`sample.toml`).

```
[[pass]]
host = server_host    ## host name of the server
path = /cock/fstore
cmd = "./fserver -db_type ycsb -model_config=
    ycsb-model.toml"

[[pass]]
host = client_host    ## host name of the client
path = /cock/fstore
cmd = "./ycsb -threads 1 -server_host
    server_host"

[[pass]]
host = master_host    ## host name of the client
path = /cock/fstore
cmd = "./master -client_config cs.toml -epoch 60
     -nclients 1"
```

Mellanox OFED.

```
$ wget latest_ofed_for_the_OS.
$ ./mlnxofedinstall -without-iser-dkms
-without-srp-dkms -without-srptools -force
```

Boost and jemalloc.

```
$ cd path_to_xstore
$ pip3 install -r requirements.txt
$ ./magic.py config -f build-config.toml
$ cmake .
$ cd deps/jemalloc
$ autoconf
$ cd path_to_xstore
$ make boost jemalloc
```

XSTORE.

```
$ make fserver ycsb micro master
```

A.4 Experiment Workflow

Launch XSTORE server.

```
$ ssh server_host
$ ./fserver -db_type ycsb -model_-
config=ycsb-model.toml
```

Launch XSTORE clients.

```
$ ssh client_host
$ ./ycsb -threads 1 -server_host server_host
```

Launch a master to collect results from clients.

```
$ ssh master_host
$ ./master -client_config cs.toml -epoch 60
-nclients 1
```

Automatic experiment workflow. Optionally, readers could use our script (`bootstrap.py`) to automate the above three steps. It takes a configuration file (e.g., Listing 1) to execute the above three steps required for the experiments. Specifically, the following command should launch the server, the clients, and the master accordingly:

```
$ ./bootstrap.py -f sample.toml
```

Table 4: The evaluating scripts to reproduce the results in §7. Note that the scripts are in the `ae_scripts` folder in our repository.

Figure	Description	Evaluating script
Fig. 10	YCSB A	`ycsba.toml`
Fig. 11	YCSB B	`ycsbb.toml`
Fig. 12	YCSB C	`ycsbc.toml`
	YCSB D	`ycsbd.toml`
	YCSB E	`ycsbe.toml`
Fig. 14c	Model expansion	`expan.toml`
Fig. 15d,e	Data distribution	`ln.toml`
Fig. 15b,c	Memory footprint	`cached_ycsbc.toml`

A.5 Evaluation and Expected Result

The experimental results mainly include throughput and meidan latency. By default, the `master` is responsible for printing the results. It should be noted that it is difficult to compare the performance results across different machines. Therefore, we only show the reported numbers on our testbed as an example here. For instance, to evaluate YCSB C on XSTORE, readers could run the script as follows:

```
$ ./bootstrap.py -f ae_scripts/ycsbc.toml
```

The `master` would print throughput (*thpt*) and median latency (*lat*) per second:

```
...
At epoch 1 thpt: 79.4M/s, ..., lat: 19.5 us
At epoch 2 thpt: 79.3M/s, ..., lat: 19.6 us
At epoch 3 thpt: 79.8M/s, ..., lat: 19.4 us
...
```

A.6 Experiment Customization

Table 4 lists the configuration files used to produce the experimental results in our paper. However, the scripts mainly target our testbed (§7.1). To execute them on other platforms, readers need to make minor changes to the scripts. The `README` in our repository provides detailed information about how to customize them for the experiments.

A.7 Notes

The source code and scripts for the artifact evaluation are used to reproduce the main results in XSTORE. To use XSTORE in your research, we recommend the `main` branch of our repository (§A.2.1), which would be maintained by members of the Institute of Parallel and Distributed Systems.

A.8 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts





CrossFS: A Cross-layered Direct-Access File System

Yujie Ren
Rutgers University
yujie.ren@rutgers.edu

Changwoo Min
Virginia Tech
changwoo@vt.edu

Sudarsun Kannan
Rutgers University
sudarsun.kannan@rutgers.edu

Abstract

We design CrossFS, a cross-layered direct-access file system disaggregated across user-level, firmware, and kernel layers for scaling I/O performance and improving concurrency. CrossFS is designed to exploit host- and device-level compute capabilities. For concurrency with or without data sharing across threads and processes, CrossFS introduces a file descriptor-based concurrency control that maps each file descriptor to one hardware-level I/O queue. This design allows CrossFS's firmware component to process disjoint access across file descriptors concurrently. CrossFS delegates concurrency control to powerful host-CPUs, which convert the file descriptor synchronization problem into an I/O queue request ordering problem. To guarantee crash consistency in the cross-layered design, CrossFS exploits byte-addressable nonvolatile memory for I/O queue persistence and designs a lightweight firmware-level journaling mechanism. Finally, CrossFS designs a firmware-level I/O scheduler for efficient dispatch of file descriptor requests. Evaluation of emulated CrossFS on storage-class memory shows up to 4.87x concurrent access gains for benchmarks and 2.32x gains for real-world applications over the state-of-the-art kernel, user-level, and firmware file systems.

1 Introduction

We have finally entered an era where storage access latency is transitioning from milliseconds to microseconds [3,52,62,68]. While modern applications strive to increase I/O parallelism, storage software bottlenecks such as system call overheads, coarse-grained concurrency control, and the inability to exploit hardware-level concurrency continues to impact I/O performance. Several kernel-level, user-level, and firmware-level file systems have been designed to benefit from CPU parallelism [65, 66], direct storage access [22, 31, 40], or computational capability in the storage hardware [33, 56]. However, these approaches are designed in isolation and fail to exploit modern, ultra-fast storage hardware.

Kernel-level file systems (Kernel-FS) satisfy *fundamental file system guarantees* such as integrity, consistency, durability, and security. Despite years of research, Kernel-FS designs continue to suffer from **three main bottlenecks**. First, applications must enter and exit the OS for performing I/O, which could increase latency by 1-4μs [31, 68]. Recently found security vulnerabilities have further amplified such costs [25, 39, 47]. Second, even state-of-the-art designs enforce unnecessary serialization (e.g., inode-level read-write lock) when accessing disjoint portions of data in a file leading to high concurrent access overheads [48]. Third, Kernel-FS designs *fail to fully exploit* storage hardware-level capabilities such as compute, thousands of I/O queues, and firmware-level scheduling, ultimately impacting I/O latency, throughput, and concurrency in I/O-intensive applications [5, 8, 9, 45, 69].

As an alternative design point, there is increasing focus towards designing user-level file systems (User-FS) for direct storage access bypassing the OS [17, 22, 31, 40, 52, 65]. However, satisfying the fundamental file system guarantees from untrusted user-level is challenging [33]. While these designs have advanced the state of the art, some designs bypass the OS only for data-plane operations (without data sharing) [17, 31, 52]. In contrast, others provide full direct access by either sidestepping or inheriting coarse-grained and suboptimal concurrency control across threads and processes [22, 40], or even compromise correctness [65]. Importantly, most User-FS designs fail to exploit the hardware capabilities of modern storage.

At the other extreme is the exploration of firmware-level file systems (Firmware-FS) that embed the file system into the device firmware for direct-access [33, 56]. The Firmware-FS acts as a central entity to satisfy fundamental file system properties. Although an important first step towards utilizing storage-level computational capability, current designs miss out on benefiting from host-level multi-core parallelism. Additionally, these designs inherit inode-centric design for request queuing, concurrency control, and scheduling, leading to poor I/O scalability.

In summary, current User-FS, Kernel-FS, and Firmware-FS

designs *lack a synergistic design across the user, the kernel, and the firmware layers*, which is critical for achieving direct storage access and scaling concurrent I/O performance without compromising fundamental file system properties.

Our Approach - Cross-layered File System. To address the aforementioned bottlenecks, we propose **CrossFS**, a cross-layered direct-access file system that provides scalability, high concurrent access throughput, and lower access latency. CrossFS achieves these goals by disaggregating the file system across the user-level, the device firmware, and the OS layer, thereby exploiting the benefits of each layer. The firmware component (FirmFS) is the heart of the file system enabling applications to directly access the storage without compromising fundamental file system properties. The FirmFS taps into storage hardware's I/O queues, computational capability, and I/O scheduling capability for improving I/O performance. The user-level library component (LibFS) provides POSIX compatibility and handles concurrency control and conflict resolution using the host-level CPUs (host-CPUs). The OS component sets up the initial interface between LibFS and FirmFS (e.g., I/O queues) and converts software-level access control to hardware security control.

Scalability. File system disaggregation alone is insufficient for achieving I/O scalability, which demands revisiting file system concurrency control, reducing journaling cost, and designing I/O scheduling that matches the concurrency control. We observe that file descriptors (and not inode) are a natural abstraction of access in most concurrent applications, where threads and processes use independent file descriptors to access/update different regions of shared or private files (for example, RocksDB maintains 3.5K open file descriptors). Hence, for I/O scalability, in CrossFS, we introduce file descriptor-based concurrency control, which allows threads or processes to update or access non-conflicting blocks of a file simultaneously.

Concurrency Control via Queue Ordering. In CrossFS, file descriptors are mapped to dedicated hardware I/O queues to exploit storage hardware parallelism and fine-grained concurrency control. All non-conflicting requests (i.e., requests to different blocks) issued using a file descriptor are added to a file descriptor-specific queue. In contrast, conflicting requests are ordered by using a single queue. This provides an opportunity for device-CPUs and FirmFS to dispatch requests concurrently with almost zero synchronization between host and device-CPUs. For conflict resolution and ordering updates to blocks across file descriptors, CrossFS uses a per-inode interval tree [7], interval tree read-write semaphore (interval tree `rw-lock`), and global timestamps for concurrency control. However, *unlike current file systems that must hold inode-level locks* until request completion, CrossFS only acquires interval tree `rw-lock` for request ordering to FD-queues. In short, CrossFS concurrency design *turns the file synchronization problem into a queue ordering problem*.

CrossFS Challenges. Moving away from an inode-centric to a file descriptor-centric design introduces CrossFS-specific challenges. First, using fewer and wimpier device-CPUs for conflict resolution and concurrency control impacts performance. Second, mapping a file descriptor to an I/O queue (a device-accessible DMA memory buffer) increases the number of queues that CrossFS must manage, potentially leading to data loss after a crash. Finally, overburdening device-CPUs for serving I/O requests across hundreds of file descriptor queues could impact performance, specifically for blocking I/O operations (e.g., `read, fsync`).

Host Delegation. To overcome the challenge of fewer (and wimpier) device-CPUs, CrossFS utilizes the cross-layered design and delegates the responsibility of request ordering to host-CPUs. The host-CPUs order data updates to files they have access to, whereas FirmFS is ultimately responsible for updating and maintaining metadata integrity, consistency, and security with POSIX-level guarantees.

Crash-Consistency and Scheduler. To handle crash consistency and protect data loss across tens and possibly hundreds of FD-queues, CrossFS uses byte-addressable, persistent NVMs as DMA'able and append-only FD-queues from which FirmFS can directly fetch requests or pool responses. CrossFS also designs low-cost data journaling for crash-consistency of firmware file system state (§4.4). Finally, for efficient scheduling of device-CPUs, CrossFS smashes traditional two-level I/O schedulers spread across the host-OS and the firmware into one FirmFS scheduler. CrossFS also equips the scheduler with policies that enhance file descriptor-based concurrency.

Evaluation of our CrossFS prototype implemented as a device-driver and emulated using Intel Optane DC memory [3] shows significant concurrent access performance gains with or without data sharing compared to state-of-the-art Kernel-FS [64, 66], User-FS [31, 40], and Firmware-FS [33] designs. The performance gains stem from reducing system calls, file descriptor-level concurrency, work division across host and device-CPUs, low-overhead journaling, and improved firmware-level scheduling. The concurrent access microbenchmarks with data sharing across threads and processes show up to 4.87X gains. The multithreaded Filebench [59] macrobenchmark workloads without data sharing show up to 3.58X throughput gains. Finally, widely used real-world applications such as RocksDB [9] and Redis [8] show up to 2.32X and 2.35X gains, respectively.

2 Background and Related Work

Modern ultra-fast storage devices provide high bandwidth (8-16 GB/s) and two orders of lower access latency (< 20μsec) [15, 67] compared to HDD storage. The performance benefits can be attributed to innovation in faster storage hardware and access interface (e.g., PCIe support), increase in storage-level compute (4-8 CPUs, 4-8 GB DRAM,

64K I/O queues) [27], fault protection equipped with capacitors [57, 58], and evolving support for storage programmability. In recent years, file systems have evolved to exploit high-performance modern storage devices. However, for applications to truly benefit from modern storage, file system support for scalable concurrency and efficient data sharing is critical [21, 26, 42]. Next, we discuss kernel, user-level, and firmware-level file system innovations and their implications.

Kernel-level File Systems (Kernel-FS). Several new kernel file systems have been designed to exploit the capabilities of modern storage devices [15, 23, 43, 64, 66]. For example, F2FS exploits multiple queues of an SSD and employs a log-structured design [43]. LightNVM moves the FTL firmware code to the host and customizes request scheduling [15]. File systems such as PMFS [23], DAX [64], and NOVA [66] exploit NVM's byte-addressability; they support block-based POSIX interface but replace block operations with byte-level loads and stores. To improve concurrent access performance, file systems such as ext4 introduce inode-level read-write semaphores (`rw-lock`) for improving read concurrency when sharing files [24]. Alternatively, user-level storage access frameworks such as SPDK use NVMe-based I/O command queues to provide direct I/O operations. However, these frameworks only support simple block operations as opposed to a POSIX interface [29].

User-level File Systems (User-FS). There is a renewed focus to bypass the OS and directly access storage hardware from a user-level library. However, managing a file system from an untrusted library introduces a myriad of challenges, which include atomicity, crash consistency, and security challenges [40, 52, 60]. Some prior designs bypass the OS for data plane operations but enter the OS for control plane operations [31, 52, 60]. In contrast, approaches such as Strata [40] and ZoFS [22] provide direct access for control and data plane operations by managing data and metadata in a user-level library. For example, Strata [40] buffers data and metadata updates to a per-process, append-only log, which is periodically committed to a shared area using a trusted file system server. More recently, ZoFS, designed for persistent memory technologies, uses virtual memory protection to secure access to a user-level buffer holding data and metadata updates. Unfortunately, all these approaches require high-overhead concurrency control and use coarse-grained locks that do not scale [31]. For example, in Strata, a process must acquire an inode lease from the trusted file system server before accessing a file [40, 60].

Firmware File Systems (Firmware-FS). After two decades of seminal work on programmable storage [17, 33, 54, 56], prior research takes a radical approach of offloading either the entire file system [33] or a part of it [17, 54] into the device firmware. The firmware approach allows applications to bypass the OS for both control and data plane operations. Firmware-FS acts as a central entity to coordinate updates to file system metadata and data without compromising crash-consistency by using the device-CPUs and the device-RAM for per-inode I/O queues. For security and permission checks, systems such as DevFS [33] rely on the host OS to update a device-level credential table with process permissions. For crash consistency, the power-loss capacitors could be used to flush in-transit updates after a system failure. Insider [56] explores the use of FPGAs for file system design, whereas other efforts have focused on using FPGAs [70] to accelerate key-value stores. Unfortunately, both DevFS and Insider handle concurrency using inode-level locks, limiting file system concurrency and scalability.

ext4-DAX	Strata	DevFS	CrossFS
21.38%	26.57%	27.06%	9.99%

Table 1: Time spent on inode-level lock.

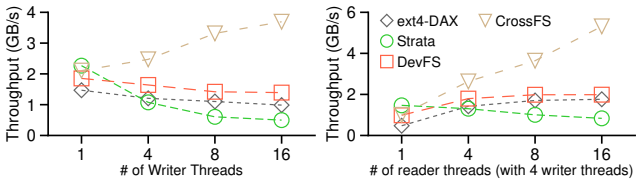
File System Scalability. Several kernel-level file system scalability designs have been explored in the past. SpanFS [32] shares files and directories across cores at a coarse granularity, requiring developers to distribute I/O. ScaleFS [13] decouples the in-memory file system from the on-disk file system and uses per-core operation logs to achieve high concurrency. FLEX [65] moves file operations to a user-level library and modifies the Linux VFS layer. Fine-grained locking mechanisms such as CST-semaphore [36] and range lock [42] are applied to current kernel-level file systems to improve concurrency. However, all the above approaches either lack direct access or require inode-level locks and fail to utilize host and device-level compute capabilities.

3 Motivation

Unfortunately, state-of-the-art Kernel-FS [64, 66], User-FS [31, 40], and Firmware-FS [33] designs suffer from three prominent limitations. First, they lack a synergistic design that could benefit from the combined use of host and device resources, such as host and device-level CPUs, and thousands of hardware I/O queues. Second, the use of an inode-centric design limits concurrent access scalability. Third, for file sharing across processes, applications must trap into an OS for control or data plane or both [60].

To briefly illustrate the poor concurrent access scalability in state-of-the-art file system designs, we conduct an experiment where readers and writers perform random-but-disjoint block accesses on a 12GB file. For our analysis, we compare User-FS Strata [40], Kernel-FS Linux ext4-DAX [64], Firmware-FS DevFS [33], and the proposed CrossFS. Figure 1a shows the aggregate write throughput when multiple writers concurrently update a shared file. The x-axis varies the writer thread count. In Figure 1b, we restrict the number of writers to *4* and increase readers in the x-axis.

Concurrent Write Performance: As shown in Figure 1a, when sharing files across concurrent writers, the throughput substantially reduces for all approaches except CrossFS. ext4-



(a) Concurrent Writers **(b)** Concurrent Readers

Figure 1: Concurrent Write and Read Throughput. (a) shows the aggregated write throughput when concurrent writers update disjoint random blocks of a 12GB file; (b) shows aggregated read throughput when there are 4-concurrent writers.

DAX and NOVA use inode-level `rw-lock`, which prevents writers from updating disjoint blocks of a file. Strata does not scale beyond four threads because it uses an inode-level mutex. Additionally, Strata uses a private NVM log to buffer data and metadata (inode) updates, and the log fills up frequently, consequently stalling write operations.

Sharing across Reader and Writer Threads: Figure 1b and Table 1 show the aggregated read throughput and the execution time spent on an inode-level `rw-lock`. The read performance does not scale (in the presence of writers) even when accessing disjoint blocks, mainly due to the inode-level `rw-lock`. For Strata, by the time readers acquire a mutex for the private log, the log contents could be flushed, forcing readers to access data using Kernel-FS. We see similar performance bottlenecks when using concurrent processes instead of threads for ext4-DAX and NOVA due to their inode-centric concurrency control. For Strata, the performance degrades further; the reader processes starve until writers flush their private log to the shared area. These issues highlight the complexities of scaling concurrent access performance in Kernel-FS and User-FS designs. The observations hold for other user-level file systems such as ZoFS [22]. Finally, as shown in Figure 1a and Figure 1b, CrossFS outperforms other file systems with its fine-grained file descriptor-based concurrency. We will next discuss the design details of CrossFS.

4 Design of CrossFS

CrossFS is a cross-layered design that disaggregates the file system to exploit the capabilities of userspace, firmware, and OS layers. CrossFS also achieves high end-to-end concurrency in the user-level library, I/O request queues, and firmware-level file system, with or without file sharing across threads and processes. Finally, CrossFS reduces system call cost, provides lightweight crash consistency, and efficient device-CPU scheduling. We observe that applications that perform conflicting updates to the same blocks of files, automatically resort to protecting file descriptors with application-level locking [2, 5, 9]. Hence, for concurrency and file sharing, CrossFS uses file descriptors as a fundamental unit of synchronization, enabling threads and processes with separate file descriptors to update disjoint blocks of files concurrently. CrossFS assigns each file descriptor a dedicated I/O queue (**FD-queue**) and adds non-conflicting (disjoint) block requests across descriptors to their independent FD-queue, whereas serializing conflicting block updates or access to a single FD-queue. Consequently, CrossFS converts the file synchronization problem to an I/O request ordering problem, enabling device-CPUs to dispatch requests across FD-queues concurrently. We outline the design principles and then follow it up by describing the details of file descriptor-based concurrency mechanism, crash consistency support, I/O scheduling, and security.

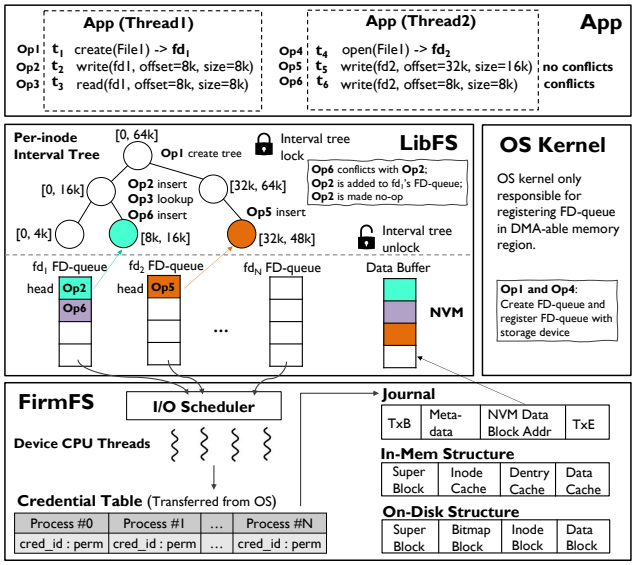


Figure 2: CrossFS Design Overview. The host-level LibFS converts POSIX I/O calls into FirmFS commands, manages FD-queues, and interval tree for checking block conflicts and ordering requests in FD-queues. FirmFS implements the file system, journaling, and scheduler and concurrently processes requests across FD-queues. The OS component is responsible for the FD-queue setup and updates the device credential table with host-level permission information. We show a running example of two instances sharing a file; Op1 to 6 show request execution with global timestamps t1 to t6. Op6 conflicts with Op2, so Op6 is added to the same FD-queue as Op2 using an interval tree.

4.1 CrossFS Design Principles

CrossFS adapts the following key design principles:

Principle 1: For achieving high performant direct-I/O, disaggregate file system across user-level, firmware, and OS layers to exploit host and device-level computing capabilities.

Principle 2: For fine-grained concurrency, align each file descriptor to an independent hardware I/O queue (FD-queue), and design concurrency control focused around the file descriptor abstraction.

Principle 3: To efficiently use device-CPUs, merge software and hardware I/O schedulers into a single firmware scheduler, and design scheduling policies to benefit from the file-descriptor design.

Principle 4: For crash consistency in a cross-layered design,

protect the in-transit user data and the file system state by leveraging persistence provided by byte-addressable NVMs.

4.2 CrossFS Layers

CrossFS enables unmodified POSIX-compatible applications to benefit from direct storage access. As shown in Figure 2, CrossFS comprises of a user-level library (LibFS), a firmware file system component (FirmFS), and an OS component.

User-space Library Component (LibFS). Applications are linked to LibFS, which intercepts POSIX I/O calls and converts them to FirmFS I/O commands. As shown in Figure 2, when opening a file, LibFS creates an FD-queue by requesting the OS to allocate a DMA'able memory region on NVM and registering the region with FirmFS. Each I/O request is added to a file descriptor-specific FD-queue when there are no block-level conflicts. In the presence of block conflicts, the conflicting requests are serialized to the same FD-queue. To identify block conflicts, in CrossFS, we use a per-inode interval tree with range-based locking. The conflict resolution using the interval tree is delegated to the host-CPU when sharing data across multiple threads, and for inter-process file sharing, interval tree updates are offloaded to FirmFS (and the device-CPUs).

Firmware File System Component (FirmFS). FirmFS is responsible for fetching and processing I/O requests from FD-queues. Internally, FirmFS's design is similar to a traditional file system with in-memory and on-disk metadata structures, including a superblock, bitmap blocks, and inode and data blocks. FirmFS also supports data and metadata journaling using a dedicated journal space on the device, as shown in Figure 2. FirmFS fetches I/O requests from FD-queues and updates in-memory metadata structures (stored in the device-level RAM). For crash-consistency, FirmFS journals the updates to the storage and then checkpoints them (§4.4). The mounting process for FirmFS is similar to a traditional file system: finding the superblock, followed by the root directory. To schedule requests from FD-queues, FirmFS also implements a scheduler (§4.5). Finally, FirmFS implements a simple slab allocator for managing device memory.

OS Component. The OS component is mainly used for setting up FD-queues by allocating DMA'able memory regions, mounting of CrossFS, and garbage collecting resources. The OS component also converts process-level access controls to device-level credentials for I/O permission checks without requiring applications to trap into the OS (§4.6).

4.3 Scaling Concurrent Access

We next discuss CrossFS's file descriptor-based concurrency design that scales without compromising correctness and sharply contrasts with prior inode-centric designs.

4.3.1 Per-File Descriptor I/O Queues

Modern NVMe devices support up to 64K I/O queues, which can be used by applications to parallelize I/O requests. To exploit this hardware-level I/O parallelism feature, CrossFS aligns each file descriptor with a dedicated I/O queue (with a configurable limit on the maximum FD-queues per inode). As shown in Figure 2, during file open (*Op1*), LibFS creates an FD-queue (I/O request + data buffer) by issuing an IOCTL call to the OS component, which reserves memory for an FD-queue, and registers the FD-queue's address with FirmFS. For handling uncommon scenarios where the number of open file descriptors could exceed the available I/O queues (e.g., 64K I/O queues in NVMe), supporting FD-queue reuse and multiplexing could be useful. For reuse, CrossFS must service pending requests in an FD-queue, and clear its data buffers. For multiplexing, CrossFS must implement a fair queue sharing policy. While CrossFS currently supports queue reuse after a file descriptor is closed, our future work will address the support for FD-queue multiplexing.

4.3.2 Concurrency Constraints

CrossFS provides correctness and consistency guarantees of a traditional kernel-level file system (e.g., ext4). We first define the constraints that arise as a part of CrossFS's file-descriptor design, and then describe how CrossFS satisfies these constraints.

- Constraint 1: Read requests (commands) entering a device at timestamp T must fetch the most recent version of data blocks, which are either buffered in FD-queues or stored in the storage. The *timestamp* refers to a globally incrementing atomic counter added to an I/O command in the FD-queues.
- Constraint 2: For concurrent writes to conflicting (same) blocks across file descriptors, the most recent write at T_j can overwrite a preceding write at T_i where $i < j$.

4.3.3 Delegating Concurrency Control to Host-CPUs

Handling conflict resolution across several threads that concurrently update a shared file could be compute-heavy. As a consequence, using fewer (or wimpier) device-CPUs could pose a scalability challenge. Hence, in CrossFS, for data sharing across multiple threads (a common scenario in most applications), we exploit our cross-layered design and delegate concurrency control to host-CPUs without impacting file system correctness. In the case of data sharing across processes, for simplicity and security, we offload concurrency control to FirmFS (discussed in §4.6). We first discuss the data structure support and then discuss CrossFS support for concurrent reader and writer threads with host-delegated concurrency control.

Request Timestamp. Before adding a request to an FD-queue, LibFS tags each I/O request with a globally incrementing hardware TSC-based timestamp. The timestamp is used to resolve ordering conflicts across writers and fetch the

most recent data for readers. The use of TSC-based timestamp for synchronization has been widely studied by prior research [35, 38, 46, 55].

Per-Inode Interval Tree. To resolve conflicts across concurrent writers and readers, we use a per-inode interval tree. An interval tree is an augmented red-black tree, indexed by interval ranges (low, high) and allows quick lookup for any overlapping or exact matching block range [7, 28].

In CrossFS, an inode-level interval tree is used for conflict resolution, i.e., to identify pending I/O requests in FD-queues that conflict with a newly issued I/O request. The OS allocates the interval tree in a DMA'able region during file creation and returns interval tree's address to LibFS. For each update request, LibFS adds a new interval tree node indexed by the request's block range. The nodes store a timestamp and a pointer to the newly added I/O request in the FD-queue. The interval tree's timestamp (TSC) is checked for following cases: (1) for writes to identify conflicting updates to block(s) across different FD-queues; (2) for reads to fetch the latest uncommitted (in-transit) updates from an FD-queue; and (3) for file commits (`fsync`). The interval tree items are deleted after FD-queue entries are committed to the storage blocks (discussed in 4.3.6).

Threads that share a file also share inode-level interval tree. For files shared within a process, interval tree is updated and accessed just by LibFS. The updates to interval tree are protected using a read-write lock (`rw-lock`), held only at the time request insertion to FD-queue.

4.3.4 Supporting Concurrent Readers and a Writer

CrossFS converts a file synchronization problem to a queue ordering problem. Hence, the first step towards concurrent write and read support for a file is to identify a file descriptor queue to which a request must be added. To allow concurrent readers and one writer to share a file, requests are ordered using a global timestamp.

For each write request, LibFS acquires an inode's interval tree `rw-lock` and atomically checks the interval tree for conflicts. A conflict exists if there are unprocessed prior FD-queue write requests that update the same block(s). After detecting a conflict, LibFS adds a global timestamp to the write request and orders it in the same FD-queue of the prior request. LibFS also updates the interval tree's node to point to the current request and releases the interval tree's `rw-lock`. For example, in Figure 2, for *Op6*, Thread 2's write to offset 8k conflicts with Thread 1's *Op2* buffered in *fd1*'s queue with an earlier timestamp. Hence, Thread 2's request is added to *fd1's* FD-queue for ordering the updates.

Note that, while prior systems acquire the inode `rw-lock` until request completion, *CrossFS acquires inode-level interval tree `rw-lock` only until a request is ordered to the FD-queue*. This substantially reduces the time to acquire interval tree `rw-lock` compared to prior inode-centric designs (see §6). To reduce the latency when reading block(s) from a file, LibFS uses the interval tree to identify conflicting writes to the same block(s) buffered in the FD-queue. If a conflict exists, LibFS (i.e., the host-CPU) returns the requested blocks from FD-queue, thereby reducing FirmFS work. For example, as shown in Figure 2, for *Thread 1's Op3*, LibFS checks the file descriptors *fd1*'s FD-queue for a preceding write to the same block and returns the read from the FD-queue. Read operations also acquire interval tree `rw-lock` before lookup, and do not race with an on-going write or a FD-queue cleanup (which also acquires interval tree `rw-lock`).

4.3.5 Supporting Concurrent Writers

CrossFS supports concurrent writers to share files without enforcing synchronization between host and device-CPUs.

Non-conflicting Concurrent Writes. When concurrent writers share a file, non-conflicting (disjoint) writes to blocks across file descriptors are added to their respective FD-queues, tagged with a global timestamp, and added to an interval tree node for the corresponding block range. In Figure 2, non-conflicting *Op2* in *fd1* and *Op5* in *fd2* are added to separate FD-queues and can be concurrently processed by FirmFS.

Conflicting Concurrent Writes. The number of conflicting blocks across concurrent writers could vary.

(1) Single-block conflict. A single-block conflict refers to a condition where a conflict occurs for one block. When a current writer updates one block (say block k) at a timestamp (say j), the write request (W_jB_k) is atomically checked against the interval tree for preceding non-committed writes. If an earlier write (say W_iB_k, where $i < j$) exists, CrossFS adds the request (W_jB_k) to the queue, marks the earlier request (W_iB_k) as a *no-op*, and updates the interval tree to point to the new request (W_jB_k), thereby avoiding an extra write operation. For example, in Figure 2, the later request *Op6* is added to the FD-queue of *Op2*, and *Op2* is made a no-op.

(2) Multi-block Write conflict. CrossFS must handle multi-block conflicts for correctness, although uncommon in real-world applications. First, when a writer's request to update one block ($W_{i+1}B_k$) conflicts with an earlier write that updates multiple blocks (say $W_iB_kB_{k+1}B_{k+2}$), the prior request cannot be made a *no-op*. Hence, LibFS adds the new request ($W_{i+1}B_k$) to the same FD-queue of the prior request ($W_iB_kB_{k+1}B_{k+2}$) and inserts a child node (sub-tree) to the interval tree (B_k, B_{k+2} range) with a pointer to the newly added FD-queue request. This technique is used even for a multi-block request that conflicts with a prior single or multi-block request or when conflicting small writes update different ranges in the single block. Although this approach would incur multiple writes for some blocks, it simplifies handling multi-block conflicts that are uncommon. For rare cases, where multiple blocks of the new request ($W_iB_kB_{k+1}B_{k+2}$) conflicts with blocks across multiple FD-queues (say W_jB_k and W_nB_k+2 with j and n in different queues), we treat these as an inode-level barrier operation, which we discuss next.

(3) Multi-block Concurrent Reads. Concurrent readers always fetch the most recent blocks for each update using the interval tree. For blocks with partial updates across requests, LibFS patches these blocks by walking through the sub-tree of an interval tree range.

Support for File Appends. Several classes of applications (e.g., RocksDB [9] evaluated in this paper) extensively use file appends for logging. In CrossFS, file appends are atomic operation. To process appends, FirmFS uses an atomic transaction to allocate data blocks and update metadata blocks (i.e., inode). We use and extend atomic transaction support for `O_APPEND` from the prior NVM-based file system [23].

4.3.6 File Commit (fsync) as a Barrier

File commit (`fsync`) operations in POSIX guarantee that a successful commit operation to a file descriptor commits all updates to a file preceding the `fsync` across all file descriptors. However, in CrossFS, requests across file descriptors are added to their own FD-queue and processed in parallel by FirmFS, which could break POSIX's `fsync` guarantee. Consider a scenario where a `fsync` request added to an empty FD-queue getting dispatched before earlier pending writes in other FD-queues. To avoid these issues, CrossFS treats file commit requests as a special **barrier** operation. The `fsync` request is tagged as a barrier operation and atomically added to *all FD-queues of an inode* by acquiring an interval tree `rw-lock`. The non-conflicting requests in FD-queues are concurrently dispatched by multiple device-CPUs to reduce the cost of a barrier. We study the performance impact of `fsync` operations in §6.

4.3.7 Metadata-heavy Operations

CrossFS handles metadata-heavy operations with a file descriptor (e.g., close, unlink, file rename) and without a file descriptor (e.g., mkdir, directory rename) differently. We next discuss the details.

Metadata Operations with a File Descriptor. Metadata-heavy operations include inode-level changes, so adding these requests to multiple FD-queues and concurrently processing them could impact correctness. Additionally, concurrent processing is prone to crash consistency bugs [18, 31, 53]. To avoid such issues, CrossFS maintains a LibFS-level pathname resolution cache (with files that were opened by LibFS). The cache maintains a mapping between file names and the list of open file descriptors and FD-queue addresses. CrossFS treats these metadata-heavy operations as barrier operations and adds them to all FD-queues of an inode after acquiring an interval tree `rw-lock`. The requests in FD-queues preceding the barrier are processed first, followed by atomically processing a metadata request in one queue and removing others.

Metadata Operations without a File Descriptor. For metadata operations without a file descriptor (e.g., mkdir), CrossFS uses a global queue. Requests in the global queue are processed in parallel with FD-queue requests (barring some operations). Similar to kernel file systems, FirmFS concurrently processes non-dependent requests fetched from the global queue without compromising ordering. However, for complex operations such as a directory rename prone to atomicity and crash consistency bugs, CrossFS uses a system-wide ordering as used in traditional file systems [48]. Our current approach is to add a barrier request across all open FD-queues in the system. However, this could incur high latency when there are hundreds of FD-queues. Thus, we only add the rename request to the global queue and maintain a list of global barrier timestamps (`barrier_TSC`). Before dispatching a request, a device-CPU checks if request's `TSC` $<$ `barrier_TSC`; otherwise, delays processing the request until all prior requests before the barrier are processed and committed. CrossFS is currently designed to scale data plane operations and our future work will explore techniques to parallelize non-dependent metadata-heavy operations.

4.3.8 Block Cache and Memory-map Support.

For in-memory caching, CrossFS uses FD-queue buffers (in NVM) as data cache for accessing recent blocks but does not yet implement a shared main memory (DRAM) cache. We evaluate the benefits of using FD-queue as data buffer in §6. Implementing a shared page cache could be beneficial for slow storage devices [23, 37]. Similarly, our future work will focus on providing memory-map (`mmap`) support in CrossFS.

4.4 Cross-Layered Crash Consistency

A cross-layered file system requires a synergistic lightweight crash consistency support at the host (LibFS) and the device (FirmFS) layers. We describe the details next.

FD-queue Crash Consistency. The FD-queues buffer I/O requests and data. For I/O-intensive applications with tens of open file descriptors, providing persistence and crash consistency for LibFS-managed FD-queues could be important. Therefore, CrossFS utilizes system-level hardware capability and uses byte-addressable persistent memory (Intel Optane DC Persistent Memory [3]) for storing FD-queues. The FD-queues are allocated in NVM as a DMA'able memory. LibFS adds requests to persistent FD-queues using a well-known append-only logging protocol for the crash consistency [55, 61], and to prevent data loss from volatile processor caches, issues persistent writes using CLWB and memory fence. A commit flag is set after a request is added to the FD-queue. After a failure, during recovery, requests with an unset commit flag are discarded. Note that the interval tree is stored in the host or device RAM and is rebuilt using the FD-queues after a crash. In the absence of NVM, CrossFS uses DRAM for FD-queues, providing the guarantees of traditional kernel file systems that can lose uncommitted writes.

Low-overhead FirmFS Crash Consistency. The FirmFS, which is the heart of the CrossFS design, provides crash consistency for the file system data and metadata state in the device memory. FirmFS implements journaling using a

REDO journal maintained as a circular log buffer to hold data and metadata log entries [23, 31]. When FirmFS dispatches an `update` request, FirmFS initiates a new transaction, appends both data and metadata (e.g., inode) entries to the log buffer, and commits the transaction. However, data journaling is expensive; hence most prior NVM file systems only enable metadata journaling [18, 23, 31]. In contrast, CrossFS provides lightweight journaling by exploiting the persistent FD-queues. Intuitively, because the I/O requests and data are buffered in persistent FD-queues, the FirmFS journal can avoid appending data to the journal and provide a reference to the FD-queue buffer. Therefore, CrossFS dedicates a physical region in NVM for storing all FD-queues and buffers using our in-house persistent memory allocator. LibFS, when adding requests (e.g., read or write) to a persistent FD-queue, tags the request with the virtual address and the relative offset from the mapped FD-queue NVM buffer. FirmFS uses the offset to find the NVM physical address and stores it alongside the journal metadata. For recovery, the physical address can be used to recover the data. Consequently, CrossFS reaps the benefits of *data+metadata* journaling at the cost of metadata-only journaling. The journal entries are checkpointed when the journal is full or after a two-second (configurable) interval similar to prior file systems [23]. After a crash, during recovery, the FirmFS metadata journal is first recovered, followed by the data in the NVM FD-queues.

4.5 Multi-Queue File-Descriptor Scheduler

In traditional systems, applications are supported by an I/O scheduler in the OS and the storage firmware. In CrossFS, applications bypass the OS and lack the OS-level I/O scheduler support. As a consequence, when the number of FD-queues increase, non-blocking operations (e.g., `write`) could bottleneck blocking operations (e.g., `read, fsync`), further exacerbated by the limited device-CPU count. To address these challenges, CrossFS exploits its cross-layered design and merges two-level I/O schedulers into a single multi-queue firmware-level scheduler (FD-queue-scheduler). The FD-queue-scheduler's design is inspired by the state-of-the-art Linux *blk-queue* scheduler that separates software and hardware queues [14]. However, unlike the *blk-queue* scheduler, the FD-queue-scheduler (and FirmFS in general) is agnostic of process and thread abstractions in the host. Therefore, FD-queue-scheduler uses FD-queues as a basic scheduling entity and builds scheduling policies that decide how to map device-CPUs to serve requests from FD-queues.

4.5.1 Scheduling Policies.

The FD-queue-scheduler currently supports a simple round-robin policy and an urgency-aware scheduling policy.

Round-robin Scheduling. The round-robin scheduling aims to provide fairness. We maintain a global list of FD-queues, and the device-CPUs iterate through the global list to pick a queue currently not being serviced and schedule an I/O request from the FD-queue's head. To reduce scheduling unfairness towards files with higher FD-queue count, the round-robin scheduler performs two-level scheduling: first to pick an unserviced inode, and then across the FD-queues of the inode.

Urgency-aware Scheduling. While the round-robin scheduler increases fairness, it cannot differentiate non-blocking (e.g., POSIX `write, pwrite` return before persisting to disk) and latency-sensitive blocking (e.g., `read, fsync`) I/O operations. In particular, non-blocking operations such as `write` incur additional block writes to update metadata and data journals in contrast to read operations. Hence, in FirmFS, we implement an *urgency-aware* scheduling policy that prioritizes blocking operations without starving non-blocking operations. The device-CPUs when iterating the FD-queue list, pick and schedule blocking requests at the head. Optionally, LibFS could also tag requests as blocking and non-blocking. To avoid starving non-blocking operations, non-blocking I/O requests from FD-queues that are either full or delayed beyond a threshold (100μsec by default, but configurable) are dispatched. Our evaluation in §6 shows that urgency-aware policy improves performance for read-heavy workloads without significantly impacting write performance.

4.6 Security and Permission Checking

CrossFS matches the security guarantees provided by traditional and state-of-the-art user-level file systems [31, 40] by satisfying the following three properties. First, in CrossFS, the filesystem's metadata is always updated by the trusted FirmFS. Second, data corruptions are restricted to files for which threads and processes have write permission. Third, for files shared across processes, CrossFS allows only legal writers to update a file's data or the user-level interval tree.

File System Permissions. CrossFS aims to perform file permission checks without trapping into the OS for data plane operations. For permission management, CrossFS relies on trusted OS and FirmFS. The FirmFS maintains a credential table that maps a unique process ID to its credentials. During the initialization of per-process LibFS, the OS generates a random (128-bit) unique ID [1, 6] for each process and updates the firmware credential table with the unique ID and the process credentials [4] and returns the unique ID to LibFS. FirmFS also maintains a FD-queue to per-process unique ID mapping internally. When LibFS adds a request to its private FD-queue, it also adds the request's unique ID. Before processing a request, FirmFS checks if a request's unique ID matches FD-queue's unique ID (stored internally in FirmFS), and if they match, the I/O request's permission (e.g., write access) is checked using the credentials in the firmware table. Any mismatch between an I/O request and FD-queue IDs are identified and not processed by FirmFS. As a consequence, accidental (unintended) writes to FD-queue could be identified by FirmFS. Further, a malicious process that succeeds in forging another process's unique ID cannot use the ID in its

own FD-queue.

FD-queue Protection. First, FD-queues are privately mapped to a process address space and cannot be updated by other processes. Second, an untrusted LibFS with access permission to a file could reorder or corrupt FD-queue requests, but cannot compromise metadata. Finally, a malicious reader could add a file update request to FD-queue, but would not be processed by FirmFS.

Interval Tree. To reduce work at the device-CPUs, CrossFS uses LibFS (and host-CPUs) to manage inode-level interval tree and concurrency control. Because the interval tree is shared across writers and readers, an issue arises where a reader process (with read-only permission to a file) could accidentally or maliciously corrupt updates in the interval tree by legal writers (e.g., inserting a truncate("file", 0)).

To overcome such security complications, for simplicity, CrossFS provides two modes of interval tree updates: (a) FirmFS mode, and (b) LibFS delegation mode. In the FirmFS mode (enabled by default), the host-CPUs add I/O requests to FD-queues, but FirmFS and device-CPUs are responsible for updating the per-inode interval tree and managing concurrency control. This mode is specifically useful for cases where files are shared across multiple processes. As a consequence, file updates (e.g., truncate("file", 0)) by a malicious process with read-only access would be invalidated by the trusted FirmFS. This approach marginally impacts performance without compromising direct-I/O. In contrast, the LibFS delegation mode (an optional mode, which can be enabled when mounting CrossFS) allows the use of host-CPUs for interval tree updates but does not allow inter-process file sharing. We observe that most I/O-intensive applications (including RocksDB analyzed in the paper) without file sharing could benefit from using host-CPUs. We evaluate the trade-offs of both designs in §6.

Security Limitations. We discuss security limitations common to most User-FS designs [22, 33], and CrossFS.

Shared Data Structure Corruption. Sharing data and data-structures using untrusted user-level library makes CrossFS and other user-level file systems vulnerable to malicious/buggy updates or even DoS attacks. For example, in CrossFS, a user-level thread could corrupt the inode interval tree, impacting data correctness. Prior byte-addressable NVM designs such as ZoFS use hardware support such as Intel MPK ([22, 50]) to isolate memory regions across the file system library and application threads. Unfortunately, the isolation only protects against accidental corruption but not malicious MPK use [19].

Denial of Service (DoS) Attacks. CrossFS and most user-level designs do not handle DoS attacks, where a malicious LibFS could lock all intervals in the interval tree, perform many small updates to blocks, or force long interval tree merges for reads to those ranges. Recent studies have shown that lock-based attacks are possible for kernel file systems too [51]. One approach in CrossFS could be to use OS-level monitors to revoke threads that hold interval tree locks beyond a threshold (e.g., Linux hard/soft-lockup detector). However, a more thorough analysis is required, which will be the focus of our future work.

5 Implementation

Due to the lack of programmable storage hardware, we implement and emulate CrossFS as a device driver. CrossFS is implemented in about 13K lines of code spread across LibFS (2K LOC), FirmFS with scheduler and journaling (9K LOC), and the OS (300 LOC) components. For storage, we use Intel Optane DC Persistent Memory attached to the memory bus. To emulate device-CPUs, we use dedicated CPU cores, but also consider related hardware metrics such as device-level CPU speed, PCIe latency, and storage bandwidth (see §6). For direct-I/O, unlike prior systems that use high-overhead IOCTLs [33], the persistent FD-queues are mapped as shared memory between LibFS and the driver's address space.

LibFS. In addition to the details discussed in §4, LibFS uses a shim library to intercept POSIX operations and convert them to FirmFS compliant commands. We extend the NVMe command format that supports simple block (read and write) operations to support file system operations (e.g., open, read, write). The I/O commands are allocated using NVM, and LibFS issues the commands and sets a doorbell flag for FirmFS to begin processing. FirmFS processes a command and sets an acknowledgment bit to indicate request completion. Finally, for each file descriptor, LibFS maintains a user-level file pointer structure with a reference to the corresponding FD-queue, the file name, the interval tree, and information such as file descriptor offset. For persisting FD-queues, as described in §4.4, data updates are appended to NVM log buffers and persisted using CLWB and memory fence instructions.

FirmFS. The device-CPUs are emulated using dedicated (kernel) threads pinned to specific CPUs, whereas FirmFS block management (superblock, bitmaps, inodes, and data block management) and journaling structures implemented by extending PMFS [23] ported to Linux 4.8.2 kernel. FirmFS extends PMFS's inode and dentry with a simplified dentry structure and dentry cache for path resolution. In our file descriptor-based design, path resolution involves mapping a path to an inode containing a list of all FD-queues. For FirmFS data + metadata journaling, we use REDO logs and substitute actual data with the physical address of data in NVM FD-queues.

OS Component. We primarily extend the OS for FD-queue setup and credential management. For FD-queue setup, we implement an IOCTL in the OS to allocate contiguous DMA-able memory pages using NVM memory, mapping them to process and FirmFS device driver address spaces. The OS

component also cleans up FD-queues after a file is closed. Next, for credential management, we modify Linux process creation mechanism in the OS to generate a per-process unique ID. The OS also updates the firmware-level credential table with this information.

6 Evaluation

We compare CrossFS with state-of-the-art User-FS, Kernel-FS, and Firmware-FS using microbenchmarks, macrobenchmarks, and real-world applications to answer the following questions.

- How effective is CrossFS in exploiting the cross-layered design and file descriptor-based concurrency?
- How effective is CrossFS's FD-queue scheduler?
- What is the impact of host configuration such as FD-queue depth and device configurations such as storage bandwidth, device-CPU frequency, and PCIe latency?
- Can CrossFS's cross-layered design scale for metadata-intensive workloads with no data sharing across threads?
- Does CrossFS benefit real-world applications?

6.1 Experimental Setup

We use a dual-socket, 64-core, 2.7GHz Intel(R) Xeon(R) Gold platform with 32GB memory and a 512GB SSD. For storage and FD-queues, we use a 512GB (4x128GB) Optane DC persistent memory with 8GB/sec read and 3.8GB/sec rand-write bandwidth [30]. To emulate the proposed cross-layered file system, we reserve and use 2GB of DRAM for maintaining FirmFS in-memory state.

Besides, to study the implications of CrossFS for different storage bandwidths, PCIe latency, and device-CPU speeds, we use a CloudLab-based Intel(R) Xeon(R) CPU E5-2660 server that allows memory bandwidth throttling. We reserve 80GB memory mounted in a NUMA node for an emulated NVDIMM and vary bandwidth between 500MB/s to 10GB/s. To emulate PCIe latency, we add a 900ns software delay [49] between the time a request is added to the host's FD-queue and the time a request is marked ready for FirmFS processing. To emulate and vary device-CPU speeds, we apply dynamic voltage frequency scaling (DVFS). We enable persistence of NVM-based FD-queues and our proposed FirmFS data + metadata journaling that uses REDO logging.

For analysis, we compare CrossFS against state-of-the-art file systems in three different categories: User-FS Strata [40] and SplitFS [31], Kernel-FS ext4-DAX [64] and NOVA (a log structure NVM file system [66]), and finally, Firmware-FS DevFS [33]. To understand the benefits of avoiding system calls, for CrossFS, we compare an IOCTL-mode implementation with kernel traps but without VFS cost (CrossFS-ioctl) and a direct-mode without both kernel traps and VFS cost (CrossFS-direct).

6.2 Microbenchmarks

We first evaluate CrossFS using two microbenchmarks and then provide an in-depth analysis of how the host-side and the device-side configurations affect CrossFS performance.

6.2.1 Concurrency Analysis

In Figure 3a and Figure 3b, we vary the number of concurrent readers in the x-axis setting the concurrent writer count to four threads. For this multithreaded micro-benchmark, we use LibFS-level interval tree updates. We compare CrossFS-ioctl and CrossFS-direct against ext4-DAX, NOVA, DevFS, SplitFS, and Strata, all using Intel Optane DC persistent memory for storage.

ext4-DAX. First, ext4-DAX, a Kernel-FS, suffers from high system call and VFS cost, and locking overheads [31,63]. The inode `rw-lock` (read-write semaphore) contention between writers and readers increases with higher thread count. For the four concurrent reader-writer configuration, the time spent on locks is around 21.38%. Consequently, reader and writer throughputs are significantly impacted (see Figure 3b).

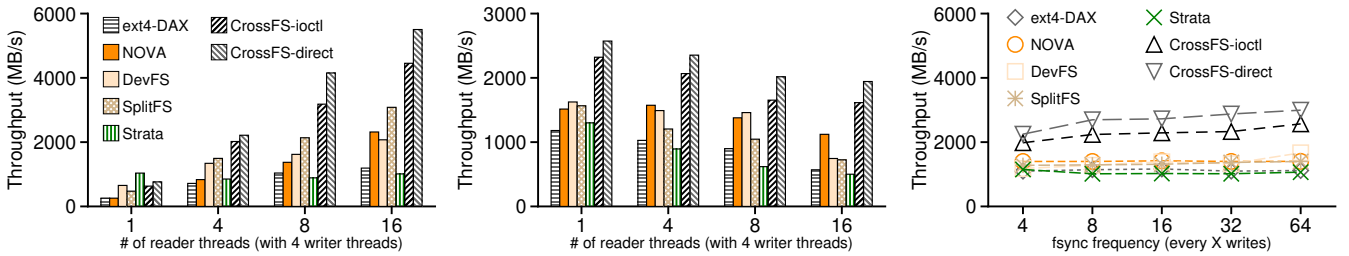
NOVA. Second, NOVA, also a Kernel-FS, exploits per-CPU file system data structures and log-structured design to improve aggregated write throughput compared to ext4-DAX. However, the use of inode-level `rw-lock` and system call costs impact write and read scalability.

DevFS. DevFS, a Firmware-FS, provides direct-access and avoids system calls, but uses per-inode I/O queues and inode-level `rw-lock`. Further, DevFS uses two levels of inode synchronization: first, when adding requests to an inode's I/O queue; second, when dispatching requests from the I/O queue for Firmware-FS processing. Consequently, the throughput does not scale with increasing thread count.

SplitFS. SplitFS, a User-FS, maps staging files to a process address space and converts data plane operations like `read()`, `write()` in the user-level library to memory loads and stores. Therefore, SplitFS avoids kernel traps for data plane operation (although metadata is updated in the kernel). As a result, SplitFS shows higher read and write throughputs over ext4-DAX, NOVA, and DevFS. However, for concurrent readers and writers, SplitFS gains are limited by inode-level locks.

Strata. Strata's low throughput is because of the following reasons: first, Strata uses an inode-level `mutex`; second, Strata uses a per-process private log (4GB by default as used in [40]). When using concurrent writers, the private log frequently fills up, *starving* readers to wait until the logs are digested to a shared area. The starvation amplifies for concurrent reader processes that cannot access private writer logs and must wait for the logs to be digested to a shared area.

CrossFS. In contrast, the proposed CrossFS-ioctl and CrossFS-direct's cross-layered designs with file descriptor concurrency allow concurrent read and write across FD-queues. Even though fewer than 1% of reads are fetched



(a) Aggregated Read Throughput **(b)** Aggregated Write Throughput **(c)** fsync Efficiency

Figure 3: Microbenchmark. Throughput of concurrent readers and 4 writers sharing a 12GB file. For CrossFS and DevFS, 4 device-CPUs are used. CrossFS-ioctl uses IOCTL commands bypassing VFS but with OS traps, whereas CrossFS-direct avoids OS traps and VFS. Figure 3a and Figure 3b are random accesses. Figure 3c shows the impact on performance when varying the commit frequency with 4 concurrent writers.

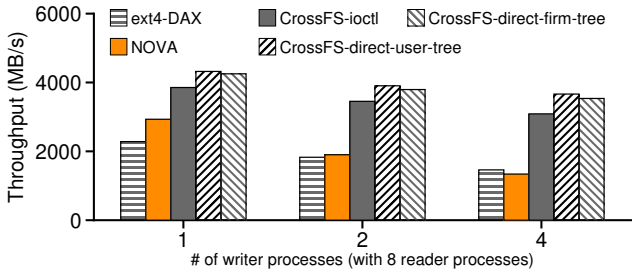


Figure 4: Process Sharing. Results show aggregated read throughput (MB/s) of 8 reader processes when sharing a file varying number of writer processes. *CrossFS-direct-user-tree* refers to using user-level interval tree, and *CrossFS-direct-firm-tree* refers to FirmFS managed interval tree, as discussed in §4.6.

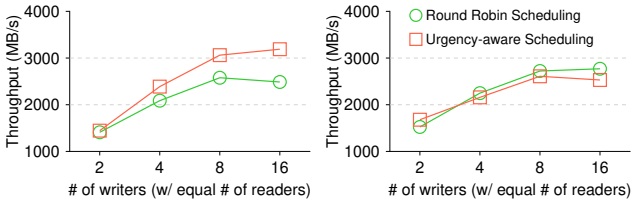
directly from the FD-queue, the performance gains are high. CrossFS-ioctl incurs kernel traps but avoids VFS overheads and low overhead journaling, resulting in 3.64X and 2.34X read performance gains over DAX and Strata, respectively. CrossFS-direct also avoids kernel traps achieving 3.38X and 4.87X write and read throughput gains over ext4-DAX, respectively. The read gains are higher because, for writes, the inode-level interval tree's `rw-lock` must be acquired (only) for FD-queue ordering. In our experiments, this accounts for only 9.9% of the execution time. Finally, as shown in Table 2, CrossFS not only improves throughput but *also reduces latency* for concurrent access.

6.2.2 Multi-Process Performance

Figure 4 shows CrossFS performance in the presence of multiple writers and readers processes. We use the same workload used earlier in Figure 3. For CrossFS-direct approach, we evaluate two cases: first, as discussed in §4.6, with CrossFS-direct-user-tree, we maintain a shared interval tree in the user-level, and LibFS updates the interval tree. While this approach reduces work for device-CPUs, a buggy or corrupted reader could accidentally or maliciously corrupt interval tree updates. In contrast, the CrossFS-direct-firm-tree approach avoids these issues by using FirmFS to update the interval tree, without impacting direct-I/O. As the figure shows, both approaches provide significant performance

	ext4-DAX	NOVA	DevFS	CrossFS-ioctl	CrossFS-direct
Readers	5.72	4.68	2.91	1.93	1.27
Writers	3.86	2.48	2.31	1.89	1.15

Table 2: Latency. Average per-thread random write and read latency (μs) with 4 concurrent readers and writers.



(a) Aggregated read throughput. **(b)** Aggregated write throughput.

Figure 5: Urgency-aware Scheduler Impact. Results show aggregated throughput for reader and writer threads. The x-axis varies the number of reader and writer threads.

gains compared to other state-of-the-art designs. Besides, CrossFS-direct-firm-tree shows only a marginal reduction in performance due to an increase in device-CPU work.

6.2.3 Commit Frequency

To study the performance of CrossFS's barrier-based commits, in Figure 3c, we evaluate `fsync` performance by running the random write benchmark with 4 concurrent writers. In the x-axis, we gradually increase the interval between successive `fsyncs`. As shown, compared to ext4-DAX, CrossFS delivers 2.05X and 2.68X performance gains for `fsyncs` issued at 4 write (worst-case) and 16 write (best-case) intervals, respectively. Although CrossFS adds a commit barrier to all FD-queues of an inode, device-CPUs can concurrently dispatch requests across FD-queues without synchronizing until the barrier completion. Additionally, CrossFS avoids system call cost for both `fsync` and write operations.

6.2.4 Urgency-aware Scheduler

CrossFS's cross-layered design smashes traditional OS-level I/O and firmware scheduler into a single firmware-level scheduler. To understand the implication of a scheduler, we evaluate two scheduling policies: (a) round-robin, which provides fairness, and (b) urgency-aware, which prioritizes

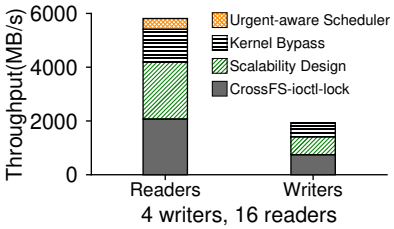


Figure 6: CrossFS Incremental Performance Breakdown. Results show aggregated throughput breakdown for 16 readers and 4 writers performing random access on a 12GB file. The baseline CrossFS approach (CrossFS-ioctl-lock) uses IOCTLs and coarse-grained inode-level lock.

blocking operations (e.g., read). We use the random-access micro-benchmark (discussed earlier) with an equal number of readers and writers performing random access on a shared 12GB file. Figure 5a and 5b show aggregated throughput for concurrent writers and readers, while varying the reader and writer count on the x-axis. First, the round-robin policy does not differentiate blocking reads and non-blocking writes; hence, it dispatches read and write requests with equal priority. Consequently, with 4 device-CPUs and the use of interval tree `rw-lock`, blocking reads are delayed. The write throughput does not scale beyond 8 threads. In contrast, CrossFS's urgency-aware scheduling prioritizes blocking reads without starving writes beyond a threshold, thereby accelerating reads by 1.22x.

6.2.5 CrossFS Performance Breakdown.

To decipher the source of CrossFS-direct performance gains, in Figure 6, we show the throughput breakdown of 4 writers, 16 readers configuration. CrossFS-ioctl-lock represents the CrossFS baseline that suffers from IOCTL (system call) cost and uses a coarse-grained inode `rw-lock`. Replacing the inode-level lock with fine-grained FD-queue concurrency (shown as Scalability Design) and eliminating system call cost (Kernel Bypass) provides significant performance benefits over the baseline. The urgency-aware scheduler improves the read throughput further.

6.2.6 Sensitivity Analysis - Host-side Configuration

We next study the performance impact of host-side configurations, which includes: (a) FD-queue depth (i.e., queue length), (b) read hits that directly fetch data from FD-queue, and (c) write conflicts with in-transit FD-queues requests. The values over the bars in Figure 7a and 7b show read hits and write conflicts (in percentage), respectively.

Read Hits. To understand the performance impact of FD-queue read hits, we mimic sequential producer-consumer I/O access patterns exhibited in multithreaded I/O-intensive applications such as Map-Reduce [20] and HPC "Cosmic Microwave Background (CMB)" [16]. The concurrent writers (producers) sequentially update specific ranges of blocks in a file, whereas the consumers sequentially read from specific ranges. Note that both producers and consumers use separate file descriptors. The consumers start at the same time as producers. In Figure 7a, for CrossFS, we vary FD-queue depths to 32 (CrossFS-direct-QD32, the default depth) and 256 (CrossFS-direct-QD256) entries. For simplicity, we only show the results for CrossFS-direct, which outperforms CrossFS-ioctl. As expected, increasing the queue depth from 32 to 256 increases the read hit rate, enabling host threads to fetch updates from FD-queues directly. Consequently, read throughput for CrossFS-direct-QD256 improves by 1.12x over CrossFS-direct-QD32, outperforming other approaches.

Write Conflicts. A consequence of increasing the queue depth is the increase in write conflicts across concurrent writers. To illustrate this, we only use concurrent writers in the above benchmark, and the writers sequentially update all blocks in a file. As shown in Figure 7b, an increase in queue-depth increases write conflicts (27% for CrossFS-direct-QD256 with 8 threads), forcing some requests to be ordered to the same FD-queue. However, this does not adversely impact the performance because of our optimized conflict resolution and fewer host-CPU stalls with 256 FD-queue entries.

6.2.7 Sensitivity Analysis - Device Configuration

A cross-layered file system could be deployed in storage devices with different bandwidths, incur PCIe latency for host and device interaction, and use wimpier CPUs. To understand CrossFS's performance sensitivity towards these device configurations, we decipher the performance by varying the storage bandwidth, adding PCIe latency, and reducing the frequency of device-CPUs.

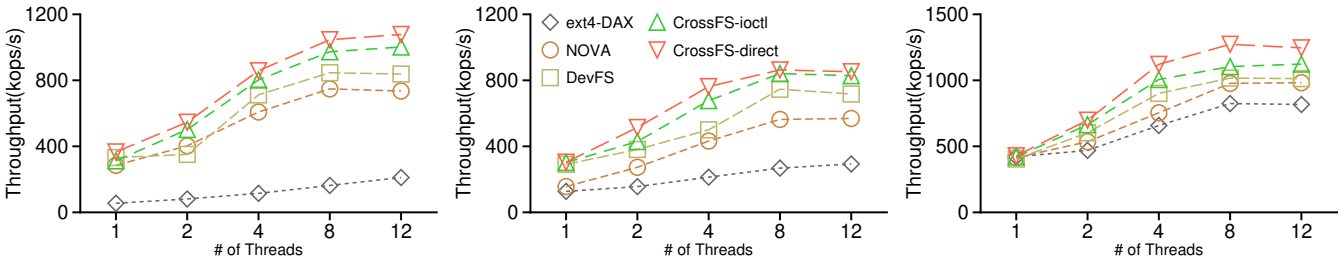
For varying the storage bandwidth, we use DRAM as a storage device and vary the storage bandwidth between 0.5GB/s to 10GB/s using thermal throttling. We use DRAM because Optane NVM cannot be thermal-throttled, and its bandwidth cannot be changed. In Figure 7c, we compare three CrossFS approaches: (1) CrossFS-noPCIeLat-HighFreq - the default approach without PCIe latency and high device-CPU frequency (2.7GHz); (2) CrossFS-PCIeLat-HighFreq - an approach that emulates *Gen 3 x8 PCIe's* latency of 900ns [49] by adding software delays between the time a request is added to a FD-queue and the time when the request is processed by FirmFS; and finally, (3) CrossFS-PCIeLat-LowFreq - an approach that reduces device-CPU frequency to 1.2GHz (the minimum frequency possible) using DVFS [41] in addition to added PCIe latency. The results show random write throughput when four concurrent writers and readers share a 12GB file. We also compare ext4-DAX, NOVA, and DevFS *without reducing CPU frequency or PCIe latency*.

At lower storage bandwidths (e.g., 500MB/s), as expected, CrossFS's gains are limited by storage bandwidth. However, even at 2GB/s bandwidth, CrossFS-noPCIeLat-HighFreq shows 1.96x gains over ext4-DAX. Next, the impact of 900ns PCIe latency (CrossFS-PCIeLat-HighFreq) is overpowered by other software overheads such as file system metadata and



(a) Read Hit Rate Impact **(b)** Write Conflicts Impact **(c)** Sensitivity to Device Configurations.

Figure 7: Performance Sensitivity towards Host and Device Configuration. Figure 7a and Figure 7b host configuration (FD-queue depth) sensitivity. Figure 7a shows read throughput with increasing in read hit rate for 32 and 256 entry FD-queue depth. Figure 7b shows write throughput and write conflict (%) when varying FD-queue depth across concurrent writers sequentially updating a file. Figure 7c shows device configuration sensitivity. The x-axis varies the storage bandwidth, and the graph shows write throughput for 4 concurrent writers when adding PCIe latency and reducing device-CPU frequency.



(a) Varmail **(b)** Fileserver **(c)** Webserver

Figure 8: Filebench Throughput. The evaluation uses 4 device-CPUs.

data management, journaling, and scheduling. While higher CPU frequency, storage bandwidth, and low PCIe latency would maximize gains when using a programmable storage, even when using 2.5x slower CPUs and 900ns PCIe latency, our cross-layered and concurrency-centric design provides 1.87x, 1.97x, 1.35x over ext4-DAX, NOVA, and DevFS that use high-frequency CPUs without PCIe latency, respectively.

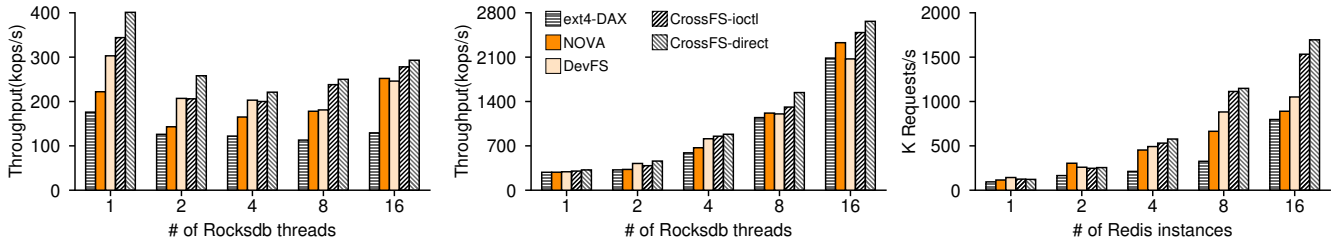
6.3 Macrobenchmark: Filebench

Does CrossFS's cross-layered design benefit multithreaded workloads without file sharing? To understand the performance, we evaluate well-known Filebench's *Varmail* (metadata-heavy), *Fileserver* (write-heavy), and *Webserver* (read-heavy) workloads that represent real-world workloads [22, 23, 66]. The workloads are metadata-intensive and perform operations such as file create, delete, directory update, which contributes to 69%, 63%, and 64% in Varmail, Fileserver, and Webserver, respectively, of the overall I/O. The data read to write ratios are 1:1, 1:2, and 10:1 in Varmail, Fileserver, and Webserver, respectively.

We compare CrossFS-ioctl and CrossFS-direct against ext4-DAX, NOVA, and DevFS. SplitFS does not yet support Filebench and RocksDB. Figure 8 shows the throughput for three workloads. The x-axis shows the throughput when increasing Filebench's thread count. Without file sharing across threads, DevFS (without system calls) performs better than NOVA and ext4-DAX. In contrast, CrossFS-direct, for *Varmail* and *Fileserver*, outperforms other approaches and provides 1.47x and 1.77x gains over NOVA, respectively. *Varmail* is metadata-intensive and (with 1:1 read-write ratio for data operations), and *fileserver* is highly write-intensive. For both these workloads, CrossFS-direct avoids system calls, reduces VFS cost, and provides fast metadata journaling at the cost of data journaling (aided by NVM-based FD-queues). With just 4 device-CPUs and a metadata I/O-queue for operations without file descriptors, CrossFS-direct gains flatten. Finally, *Webserver* has a significantly higher read ratio. While CrossFS-direct improves performance, the throughput gains (1.71x) are restricted. First, blocking reads stress the available four device-CPUs. Second, we notice OS scheduler overheads as a side-effect of emulating device-CPUs with Linux kernel threads. The kernel threads in Linux periodically check and yield to the OS scheduler if necessary (i.e., `need_resched()`); the periodic yields negatively impact blocking random read operations for CrossFS and DevFS.

6.4 Real-World Applications

RocksDB. RocksDB [9] is a widely used LSM-based NoSQL database used as a backend in several production systems and frameworks [11, 12, 44]. RocksDB is designed to exploit SSD's parallel bandwidth and multicore parallelism. As discussed earlier, we observe around 40% of I/O accesses to shared files across RocksDB's foreground threads that share



(a) Rocksdb random writes **(b)** Rocksdb random reads **(c)** Redis SET benchmark

Figure 9: Application throughput. Figure 9a and Figure 9b show random write and read performance for RocksDB by varying threads in *DBbench* workload. RocksDB internally creates background compaction threads. Figure 9c shows the *SET* benchmark for Redis by varying the number of Redis instances.

log files and background threads that compact in-memory data across LSM levels in the SST (string sorted) files [34]. However, conflicting block updates are negligible. In Figure 9a and Figure 9b, we vary the number of application (foreground) threads along the x-axis and set the device-CPU count to four. We compare ext4-DAX, NOVA, DevFS, CrossFS-ioctl, and CrossFS-direct's throughput for random write and read operations. Note that RocksDB, by default, uses three background parallel compaction threads, which increases with increasing SST levels [10]. We use widely used *DBbench* [2], with 100B keys, 1KB values, and a 32GB database size. We set RocksDB's memory buffer to 128MB [44].

CrossFS-direct and CrossFS-ioctl show up to 2.32x and 1.15x write and read throughput gains over ext4-DAX, respectively. The read and write performance benefits over DevFS are 1.33x and 1.21x, respectively. We attribute these gains to the following reasons. First, RocksDB threads do not update the same block (lower than 0.1% conflicts); hence, unlike other approaches, CrossFS-ioctl and CrossFS-direct avoid inode-level locks. Second, both CrossFS-ioctl and CrossFS-direct avoid VFS overheads. Additionally, CrossFS-direct also avoids system call costs. When increasing application threads, the burden on four device-CPUs increases, impacting performance for the 16 application thread configuration. The blocking reads are impacted due to the use of kernel threads for device-CPU emulation (see §6.3).

Redis. Redis is a widely used storage-backed in-memory key-value store [8], which logs operations to append-only-files (AOF) and checkpoints in-memory key-values asynchronously to backup files called RDB [40]. We run multiple Redis instances and the instances do not share AOF or RDB files. We use background write mode for Redis instances that immediately persist key-value updates to the disk. Figure 9c shows the Redis performance.

First, when increasing Redis instances, the number of concurrent writers increases. The server and the client (benchmark) instances run as separate processes, which increases inter-process communication, system call, and VFS costs. Although instances do not share files, CrossFS-direct provides considerable performance gains mostly stemming from direct storage access, avoiding VFS overheads, and lowering journaling cost. Consequently, CrossFS provides 2.35x higher throughput over ext4-DAX.

7 Conclusion

This paper proposes CrossFS, a cross-layered file system design that provides high-performance direct-I/O and concurrent access across threads and applications with or without data sharing. Four key ingredients contribute to CrossFS's high-performant design. First, our cross-layered approach exploits hardware and software resources spread across the untrusted user-level library, the trusted OS, and the trusted device firmware. Second, the fine-grained file descriptor concurrency design converts a file synchronization problem to the I/O queue ordering problem, which ultimately scales concurrent access. Third, our lightweight data + metadata journaling aided by NVM reduces crash consistency overheads. Finally, our unified firmware-level scheduler complements the file descriptor design, reducing I/O latency for blocking operations. Our detailed evaluation of CrossFS against state-of-the-art kernel-level, user-level, and firmware-level file system shows up to 4.87x, 3.58x, 2.32x gains on microbenchmarks, macrobenchmarks, and applications.

Acknowledgements

We thank the anonymous reviewers and Emmett Witchel (our shepherd) for their insightful comments and feedback. We thank the members of the Rutgers Systems Lab for their valuable input. This material was supported by funding from NSF grant CNS-1910593. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF.

References

[1] A Universally Unique IDentifier (UUID) URN Namespace. https://www.ietf.org/rfc/rfc4122.txt.

[2] Google LevelDB . http://tinyurl.com/osqd7c8.

[3] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[4] Linux Credentials. https://www.kernel.org/doc/html/latest/security/credentials.html.

[5] MySQL. https://www.mysql.com/.

[6] OpenSSL. https://www.openssl.org/docs/man1.1.1/man1/openssl-genrsa.html.

[7] rbtree based interval tree as a prio_tree replacement. https://lwn.net/Articles/509994/.

[8] Redis. http://redis.io/.

[9] RocksDB. http://rocksdb.org/.

[10] RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide/.

[11] Tensor Flow. https://www.tensorflow.org/.

[12] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, New York, NY, USA, 2017. ACM.

[14] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, New York, NY, USA, 2013. ACM.

[15] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-channel SSD Subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, Santa clara, CA, USA, 2017.

[16] J. Borrill, J. Carter, L. Oliker, and D. Skinner. Itegrated performance monitoring of a cosmology application on leading HEC platform. In *Proceedings of 2005 International Conference on Parallel Processing (ICPP'05)*, pages 119–128, 2005.

[17] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. *SIGARCH Comput. Archit. News*, 40(1), March 2012.

[18] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[19] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1409–1426. USENIX Association, 2020.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[21] Yang Deng, Arun Ravindran, and Tao Han. Edge Datastore for Distributed Vision Analytics: Poster. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 29:1–29:2, New York, NY, USA, 2017. ACM.

[22] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[24] Jake Edge. VFS parallel lookups. https://lwn.net/Articles/685108/.

[25] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

[26] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM.

[27] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 127–144, New York, NY, USA, 2017. ACM.

[28] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, Renton, WA, July 2019. USENIX Association.

[29] Intel. Storage Performance Development Kit. http://www.spdk.io/.

[30] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[31] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494âĂŞ–508, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 249–261, Berkeley, CA, USA, 2015. USENIX Association.

[33] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 241–255, Berkeley, CA, USA, 2018. USENIX Association.

[34] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[35] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 34:1–34:15. ACM, 2018.

[36] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 603–615, Berkeley, CA, USA, 2017. USENIX Association.

[37] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association.

[38] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 779–792. ACM, 2019.

[39] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[40] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[41] Etienne Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, page 1âĂŞ8, USA, 2010. USENIX Association.

[42] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write Optimization of Log-structured Flash File System for Parallel I/O on Manycore Servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 21–32, New York, NY, USA, 2019. ACM.

[43] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015.

[44] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[46] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 21–35. ACM, 2017.

[47] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[48] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 71–85. USENIX Association, 2016.

[49] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM'18, pages 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[50] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[51] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding Scheduler Subversion Using Scheduler-Cooperative Locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

[53] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, Santa clara, CA, USA, 2017.

[54] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, San Diego, California, 2008.

[55] Madhava Krishnan Ramanathan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with Timestone. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, pages 335–349. ACM, 2020.

[56] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[57] Samsung. NVMe SSD 960 Polaris Controller. http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/NVMe_SSD_960_PRO_EVO_Brochure.pdf.

[58] Y. Son, J. Choi, J. Jeon, C. Min, S. Kim, H. Y. Yeom, and H. Han. SSD-Assisted Backup and Recovery for Database Systems. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 285–296, April 2017.

[59] Tarasov Vasily. Filebench. https://github.com/filebench/filebench.

[60] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014.

[61] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, Newport Beach, California, USA, 2011.

[62] Michael Wei, Matias Bjørling, Philippe Bonnet, and Steven Swanson. I/O Speculation for the Microsecond Era. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, Philadelphia, PA, 2014.

[63] Zev Weiss, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. DenseFS: a Cache-Compact Filesystem. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[64] Matthew Wilcox and Ross Zwisler. Linux DAX. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[65] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, Santa Clara, CA, 2016.

[67] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, Haifa, Israel, 2015.

[68] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, San Jose, CA, 2012.

[69] Yongen Yu, Douglas H. Rudd, Zhiling Lan, Nickolay Y. Gnedin, Andrey V. Kravtsov, and Jingjin Wu. Improving Parallel IO Performance of Cell-based AMR Cosmology Applications. *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 933–944, 2012.

[70] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 225–237, Santa Clara, CA, February 2020. USENIX Association.



From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees

Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth†,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

University of Wisconsin – Madison † *Microsoft Gray Systems Lab*

Abstract. We introduce BOURBON, a log-structured merge (LSM) tree that utilizes machine learning to provide fast lookups. We base the design and implementation of BOURBON on empirically-grounded principles that we derive through careful analysis of LSM design. BOURBON employs greedy piecewise linear regression to learn key distributions, enabling fast lookup with minimal computation, and applies a cost-benefit strategy to decide when learning will be worthwhile. Through a series of experiments on both synthetic and real-world datasets, we show that BOURBON improves lookup performance by 1.23×-1.78× as compared to state-of-the-art production LSMs.

1 Introduction

Machine learning is transforming how we build computer applications and systems. Instead of writing code in the traditional algorithmic mindset, one can instead collect the proper data, train a model, and thus implement a robust and general solution to the task at hand. This data-driven, empirical approach has been called "Software 2.0" [26], hinting at a world where an increasing amount of the code we deploy is realized in this manner; a number of landmark successes over the past decade lend credence to this argument, in areas such as image [32] and speech recognition [24], machine translation [46], game playing [44], and many other areas [7,15,17].

One promising line of work, for using ML to improve core systems is that of the "learned index" [31]. This approach applies machine learning to supplant the traditional index structure found in database systems, namely the ubiquitous B-Tree [9]. To look up a key, the system uses a learned function that predicts the location of the key (and value); when successful, this approach can improve lookup performance, in some cases significantly, and also potentially reduce space overhead. Since this pioneering work, numerous follow ups [13, 20, 30] have been proposed that use better models, better tree structures, and generally improve how learning can reduce tree-based access times and overheads.

However, one critical approach has not yet been transformed in this "learned" manner: the Log-structured Merge Tree (LSM) [37, 39, 42]. LSMs were introduced in the late '90s, gained popularity a decade later through work at Google on BigTable [8] and LevelDB [22], and have become widely used in industry, including in Cassandra [33], RocksDB [18], and many other systems [21,38]. LSMs have many positive properties as compared to B-trees and their cousins, including high insert performance [11,37,40].

In this paper, we apply the idea of the learned index to LSMs. A major challenge is that while learned indexes are primarily tailored for read-only settings, LSMs are optimized for writes. Writes cause disruption to learned indexes because models learned over existing data must now be updated to accommodate the changes; the system thus must re-learn the data repeatedly. However, we find that LSMs are well-suited for learned indexes. For example, although writes modify the LSM, most portions of the tree are immutable; thus, learning a function to predict key/value locations can be done once, and used as long as the immutable data lives. However, many challenges arise. For example, variable key or value sizes make learning a function to predict locations more difficult, and performing model building too soon may lead to significant resource waste.

Thus, we first study how an existing LSM system, WiscKey [37], functions in great detail (§3). We focus on WiscKey because it is a state-of-the-art LSM implementation that is significantly faster than LevelDB and RocksDB [37]. Our analysis leads to many interesting insights from which we develop five *learning guidelines*: a set of rules that aid an LSM system to successfully incorporate learned indexes. For example, while it is useful to learn the stable, low levels in an LSM, learning higher levels can yield benefits as well because lookups must always search the higher levels. Next, not all files are equal: some files even in the lower levels are very short-lived; a system must avoid learning such files, or resources can be wasted. Finally, workload- and data-awareness is important; based on the workload and how the data is loaded, it may be more beneficial to learn some portions of the tree than others.

We apply these learning guidelines to build BOURBON, a

learned-index implementation of WiscKey (§4). BOURBON uses piece-wise linear regression, a simple but effective model that enables both fast training (i.e., learning) and inference (i.e., lookups) with little space overhead. BOURBON employs *file learning*: models are built over files given that an LSM file, once created, is never modified in-place. BOURBON implements a cost-benefit analyzer that dynamically decides whether or not to learn a file, reducing unnecessary learning while maximizing benefits. While most of the prior work on learned indexes [13, 20, 31] has made strides in optimizing stand-alone data structures, BOURBON integrates learning into a production-quality system that is already highly optimized. BOURBON's implementation adds around 5K LOC to WiscKey (which has ∼20K LOC).

We analyze the performance of BOURBON on a range of synthetic and real-world datasets and workloads (§5). We find that BOURBON reduces the indexing costs of WiscKey significantly and thus offers 1.23× – 1.78× faster lookups for various datasets. Even under workloads with significant write load, BOURBON speeds up a large fraction of lookups and, through cost-benefit, avoids unnecessary (early) model building. Thus, BOURBON matches the performance of an aggressive-learning approach but performs model building more judiciously. Finally, most of our analysis focuses on the case where fast lookups will make the most difference, namely when the data resides in memory (i.e., in the file-system page cache). However, we also experiment with BOURBON when data resides on a fast storage device (an Optane SSD) or when data does not fit entirely in memory, and show that benefits can still be realized.

This paper makes four contributions. We present the first detailed study of how LSMs function internally with learning in mind. We formulate a set of guidelines on how to integrate learned indexes into an LSM (§3). We present the design and implementation of BOURBON which incorporates learned indexes into a real, highly optimized, production-quality LSM system (§4). Finally, we analyze BOURBON's performance in detail, and demonstrate its benefits (§5).

2 Background

We first describe log-structured merge trees and explain how data is organized in LevelDB. Next, we describe WiscKey, a modified version of LevelDB that we adopt as our baseline. We then provide a brief background on learned indexes.

2.1 LSM and LevelDB

An LSM tree is a persistent data structure used in key-value stores to support efficient inserts and updates [39]. Unlike B-trees that require many random writes to storage upon updates, LSM trees perform writes sequentially, thus achieving high write throughput [39].

An LSM organizes data in multiple *levels*, with the size of each level increasing exponentially. Inserts are initially buffered in an in-memory structure; once full, this structure

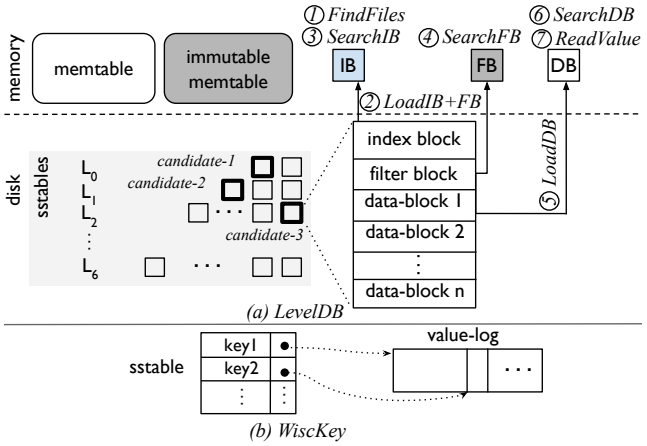


Figure 1: **LevelDB and WiscKey.** *(a) shows how data is organized in LevelDB and how a lookup is processed. The search in in-memory tables is not shown. The candidate sstables are shown in bold boxes. (b) shows how keys and values are separated in WiscKey.*

is merged with the first level of on-disk data. This procedure resembles merge-sort and is referred to as compaction. Data from an on-disk level is also merged with the successive level if the size of the level exceeds a limit. Note that updates do not modify existing records in-place; they follow the same path as inserts. As a result, many versions of the same item can be present in the tree at a time. Throughout this paper, we refer to the levels that contain the newer data as *higher* levels and the older data as *lower* levels.

A lookup request must return the latest version of an item. Because higher levels contain the newer versions, the search starts at the topmost level. First, the key is searched for in the in-memory structure; if not found, it is searched for in the on-disk tree starting from the highest level to the lowest one. The value is returned once the key is found at a level.

LevelDB [22] is a widely used key-value store built using LSM. Figure 1(a) shows how data is organized in LevelDB. A new key-value pair is first written to the *memtable*; when full, the memtable is converted into an immutable table which is then compacted and written to disk sequentially as *sstables*. The sstables are organized in seven levels (L_0 being the highest level and L_6 the lowest) and each sstable corresponds to a file. LevelDB ensures that key ranges of different sstables at a level are disjoint (two files will not contain overlapping ranges of keys); L_0 is an exception where the ranges can overlap across files. The amount of data at each level increases by a factor of ten; for example, the size of L_1 is 10MB, while L_6 contains several 100s of GBs. If a level exceeds its size limit, one or more sstables from that level are merged with the next level; this is repeated until all levels are within their limits.

Lookup steps. Figure 1(a) also shows how a lookup request for key k proceeds. ① *FindFiles*: If the key is not found in the in-memory tables, LevelDB finds the set of candidate sstable files that may contain k. A key in the worst case

may be present in all L_0 files (because of overlapping ranges) and within one file at each successive level. ② *LoadIB+FB*: In each candidate sstable, an index block and a bloom-filter block are first loaded from the disk. ③ *SearchIB*: The index block is binary searched to find the data block that may contain k. ④ *SearchFB*: The filter is queried to check if k is present in the data block. ⑤ *LoadDB*: If the filter indicates presence, the data block is loaded. ⑥ *SearchDB*: The data block is binary searched. ⑦ *ReadValue*: If the key is found in the data block, the associated value is read and the lookup ends. If the filter indicates absence or if the key is not found in the data block, the search continues to the next candidate file. Note that blocks are not always loaded from the disk; index and filter blocks, and frequently accessed data blocks are likely to be present in memory (i.e., file-system cache).

We refer to these search steps at a level that occur as part of a single lookup as an *internal lookup*. A single lookup thus consists of many internal lookups. A *negative internal lookup* does not find the key, while a *positive internal lookup* finds the key and is thus the last step of a lookup request.

We differentiate indexing steps from data-access steps; indexing steps such as *FindFiles*, *SearchIB*, *SearchFB*, and *SearchDB* search through the files and blocks to find the desired key, while data-access steps such as *LoadIB+FB*, *LoadDB*, and *ReadValue* read the data from storage. Our goal is to reduce the time spent in indexing.

2.2 WiscKey

In LevelDB, compaction results in large write amplification because *both* keys and values are sorted and rewritten. Thus, LevelDB suffers from high compaction overheads, affecting foreground workloads.

WiscKey [37] (and Badger [1]) reduces this overhead by storing the values separately; the sstables contain only keys and pointers to the values as shown in Figure 1(b). With this design, compaction sorts and writes only the keys, leaving the values undisturbed, thus reducing I/O amplification and overheads. WiscKey thus performs significantly better than other optimized LSM implementations such as LevelDB and RocksDB. Given these benefits, we adopt WiscKey as the baseline for our design. Further, WiscKey's key-value separation enables our design to handle variable-size records; we describe how in more detail in §4.2.

The write path of WiscKey is similar to that of LevelDB except that values are written to a *value log*. A lookup in WiscKey also involves searching at many levels and a final read into the log once the target key is found. The size of WiscKey's LSM tree is much smaller than LevelDB because it does not contain the values; hence, it can be entirely cached in memory [37]. Thus, a lookup request involves multiple searches in the in-memory tree, and the *ReadValue* step performs one final read to retrieve the value.

2.3 Optimizing Lookups in LSMs

Performing a lookup in LevelDB and WiscKey requires searching at multiple levels. Further, within each sstable, many blocks are searched to find the target key. Given that LSMs form the basis of many embedded key-value stores (e.g., LevelDB, RocksDB [18]) and distributed storage systems (e.g., BigTable [8], Riak [38]), optimizing lookups in LSMs can have huge benefits.

A recent body of work, starting with learned indexes [31], makes a case for replacing or augmenting traditional index structures with machine-learning models. The key idea is to train a model (such as linear regression or neural nets) on the input so that the model can predict the position of a record in the sorted dataset. The model can have inaccuracies, and thus the prediction has an associated error bound. During lookups, if the model-predicted position of the key is correct, the record is returned; if it is wrong, a local search is performed within the error bound. For example, if the predicted position is *pos* and the minimum and maximum error bounds are δ_{min} and δ_{max}, then upon a wrong prediction, a local search is performed between $pos - \delta_{min}$ and $pos + \delta_{max}$.

Learned indexes can make lookups significantly faster. Intuitively, a learned index turns a $O(log\text{-}n)$ lookup of a B-tree into a $O(1)$ operation. Empirically, learned indexes provide $1.5\times - 3\times$ faster lookups than B-trees [31]. Given these benefits, we ask the following questions: *can learned indexes for LSMs make lookups faster? If yes, under what scenarios?*

Traditional learned indexes do not support updates because models learned over the existing data would change with modifications [13, 20, 31]. However, LSMs are attractive for their high performance in write-intensive workloads because they perform writes only sequentially. Thus, we examine: *how to realize the benefits of learned indexes while supporting writes for which LSMs are optimized?* We answer these two questions next.

3 Learned Indexes: a Good Match for LSMs?

In this section, we first analyze if learned indexes could be beneficial for LSMs and examine under what scenarios they can improve lookup performance. We then provide our intuition as to why learned indexes might be appropriate for LSMs even when allowing writes. We conduct an in-depth study based on measurements of how WiscKey functions internally under different workloads to validate our intuition. From our analysis, we derive a set of learning guidelines.

3.1 Learned Indexes: Beneficial Regimes

A lookup in LSM involves several indexing and data-access steps. Optimized indexes such as learned indexes can reduce the overheads of indexing but cannot reduce data-access costs. In WiscKey, learned indexes can thus potentially reduce the costs of indexing steps such as *FindFiles*, *SearchIB*, and *SearchDB*, while data-access costs (e.g., *ReadValue*)

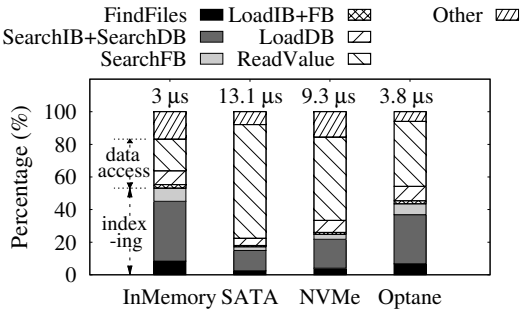


Figure 2: **Lookup Latency Breakdown.** *The figure shows the breakdown of lookup latency in WiscKey. The first bar shows the case when data is cached in memory. The other three bars show the case where the dataset is stored on different types of SSDs. We perform 10M random lookups on the Amazon Reviews dataset [5]; the figure shows the breakdown of the average latency (shown at the top of each bar). The indexing portions are shown in solid colors; data access and other portions are shown in patterns.*

cannot be significantly reduced. As a result, learned indexes can improve overall lookup performance if indexing contributes to a sizable portion of the total lookup latency. We identify scenarios where this is the case.

First, when the dataset or a portion of it is cached in memory, data-access costs are low, and so indexing costs become significant. Figure 2 shows the breakdown of lookup latencies in WiscKey. The first bar shows the case when the dataset is cached in memory; the second bar shows the case where the data is stored on a flash-based SATA SSD. With caching, data-access and indexing costs contribute almost equally to the latency. Thus, optimizing the indexing portion can reduce lookup latencies by about 2×. When the dataset is not cached, data-access costs dominate and thus optimizing indexes may yield smaller benefits (about 20%).

However, learned indexes are not limited to scenarios where data is cached in memory. They offer benefit on fast storage devices that are currently prevalent and can do more so on emerging faster devices. The last three bars in Figure 2 show the breakdown for three kinds of devices: flash-based SSDs over SATA and NVMe, and an Optane SSD. As the device gets faster, lookup latency (as shown at the top) decreases, but the fraction of time spent on indexing increases. For example, with SATA SSDs, indexing takes about 17% of the total time; in contrast, with Optane SSDs, indexing takes 44% and thus optimizing it with learned indexes can potentially improve performance by 1.8×. More importantly, the trend in storage performance favors the use of learned indexes. With storage performance increasing rapidly and emerging technologies like 3D Xpoint memory providing very low access latencies, indexing costs will dominate and thus learned indexes will yield increasing benefits.

Summary. Learned indexes could be beneficial when the database or a portion of it is cached in memory. With fast storage devices, regardless of caching, indexing contributes to a significant fraction of the lookup time; thus, learned indexes can prove useful in such cases. With storage devices getting faster, learned indexes will be even more beneficial.

3.2 Learned Indexes with Writes

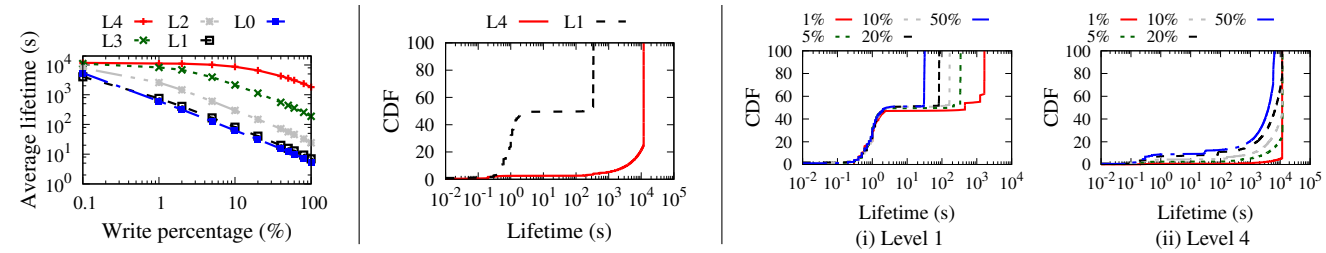
Learned indexes provide higher lookup performance compared to traditional indexes for read-only analytical workloads. However, a major drawback of learned indexes (as described in [31]) is that they do not support modifications such as inserts and updates [13, 20]. The main problem with modifications is that they alter the data distribution and so the models must be re-learned; for write-heavy workloads, models must be rebuilt often, incurring high overheads.

At first, it may seem like learned indexes are not a good match for write-heavy situations for which LSMs are optimized. However, we observe that the design of LSMs fits well with learned indexes. Our key realization is that although updates can change portions of the LSM tree, a large part remains immutable. Specifically, newly modified items are buffered in the in-memory structures or present in the higher levels of the tree, while stable data resides at the lower levels. Given that a large fraction of the dataset resides in the stable, lower levels, lookups to this fraction can be made faster with no or few re-learnings. In contrast, learning in higher levels may be less beneficial: they change at a faster rate and thus must be re-learned often.

We also realize that the immutable nature of sstable files makes them an ideal unit for learning. Once learned, these files are never updated and thus a model can be useful until the file is replaced. Further, the data within an sstable is sorted; such sorted data can be learned using simple models. A level, which is a collection of many immutable files, can also be learned as a whole using simple models. The data in a level is also sorted: the individual sstables are sorted, and there are no overlapping key ranges across sstables.

We next conduct a series of in-depth measurements to validate our intuitions. Our experiments confirm that while a part of our intuition is indeed true, there are some subtleties (for example, in learning files at higher levels). Based on these experimental results, we formulate a set of *learning guidelines*: a few simple rules that an LSM that applies learned indexes should follow.

Experiments: goal and setup. The goal of our experiments is to determine how long a model will be useful and how often it will be useful. A model built for a sstable file is useful as long as the file exists; thus, we first measure and analyze sstable lifetimes. How often a model will be used is determined by how many internal lookups it serves; thus, we next measure the number of internal lookups to each file. Since models can also be built for entire levels, we finally measure level lifetimes as well. To perform our analysis, we run workloads with varying amounts of writes and reads, and measure the lifetimes and number of lookups. We conduct our experiments on WiscKey, but we believe our results are



(a) Average lifetimes with varying write % (b) Lifetime distribution with 5% writes (c) Lifetime distributions with varying write %

Figure 3: **SSTable Lifetimes.** *(a) shows the average lifetime of sstable files in levels L_4 to L_0. (b) shows the distribution of lifetimes of sstables in L_1 and L_4 with 5% writes. (c) shows the distribution of lifetimes of sstables for different write percentages in L_1 and L_4.*

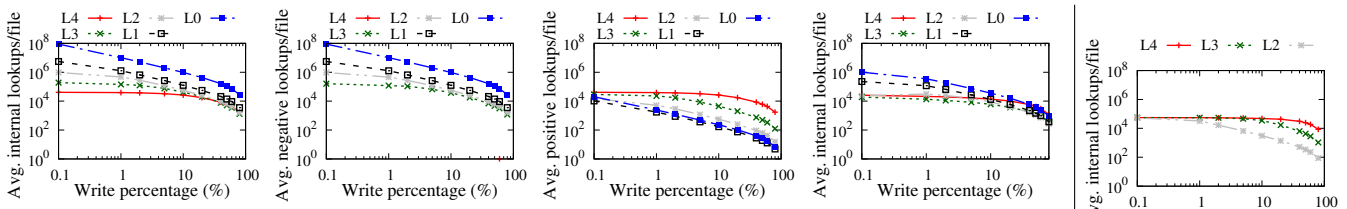


Figure 4: **Number of Internal Lookups Per File.** *(a)(i) shows the average internal lookups per file at each level for a randomly loaded dataset. (b) shows the same for sequentially loaded dataset. (a)(ii) and (a)(iii) show the negative and positive internal lookups for the randomly loaded case. (a)(iv) shows the positive internal lookups for the randomly loaded case when the workload distribution is Zipfian.*

applicable to most LSM implementations. We first load the database with 256M key-value pairs. We then run a workload with a single rate-limited client that performs 200M operations, a fraction of which are writes. Our workload chooses keys uniformly at random.

Lifetime of SSTables. To determine how long a model will be useful, we first measure and analyze the lifetimes of sstables. To do so, we track the creation and deletion times of all sstables. For files created during the load phase, we assign the workload-start time as their creation time; for other files, we record the actual creation times. If the file is deleted during the workload, then we calculate its exact lifetime. However, some files are not deleted by the end of the workload and we must estimate their lifetimes.[†]

Figure 3(a) shows the average lifetime of sstable files at different levels. We make three main observations. First, the average lifetime of sstable files at lower levels is greater than that of higher levels. Second, at lower percentages of writes, even files at higher levels have a considerable lifetime; for example, at 5% writes, files at L_0 live for about 2 minutes on an average. Files at lower levels live much longer; files at L_4 live about 150 minutes. Third, although the average lifetime of files reduces with more writes, even with a high amount of writes, files at lower levels live for a long period. For instance, with 50% writes, files at L_4 live for about 60 minutes. In contrast, files at higher level live only for a few seconds; for example, an L_0 file lives only about 10 seconds.

We now take a closer look at the lifetime distribution. Figure 3(b) shows the distributions for L_1 and L_4 files with 5% writes. We first note that some files are very short-lived, while some are long-lived. For example, in L_1, the lifetime of about 50% of the files is only about 2.5 seconds. If files cross this threshold, they tend to live for much longer times; almost all of the remaining L_1 files live over five minutes.

Surprisingly, even at L_4, which has a higher average lifetime for files, a few files are very short-lived. We observe that about 2% of L_4 files live less than a second. We find that there are two reasons why a few files live for a very short time. First, compaction of a L_i file creates a new file in L_{i+1} which is again immediately chosen for compaction to the next level. Second, compaction of a L_i file creates a new file in L_{i+1}, which has overlapping key ranges with the next file that is being compacted from L_i. Figure 3(c) shows that this pattern holds for other percentages of writes too. We observed that this holds for other levels as well. From the above observations, we arrive at our first two learning guidelines.

Learning guideline - 1: Favor learning files at lower levels. Files at lower levels live for a long period even for high write percentages; thus, models for these files can be used for a long time and need not be rebuilt often.

Learning guideline - 2: Wait before learning a file. A few

[†]If the files are created during load, we assign the workload duration as their lifetimes. If not, we estimate the lifetime of a file based on its creation time (c) and the total workload time (w); the lifetime of the file is at least $w-c$. We thus consider the lifetime distribution of other files that have a lifetime of at least $w-c$. We then pick a random lifetime in this distribution and assign it as this file's lifetime.

files are very short-lived, even at lower levels. Thus, learning must be invoked only after a file has lived up to a threshold lifetime after which it is highly likely to live for a long time.

Internal Lookups at Different Levels. To determine how many times a model will be used, we analyze the number of lookups served by the sstable files. We run a workload and measure the number of lookups served by files at each level and plot the average number of lookups per file at each level. Figure 4(a) shows the result when the dataset is loaded in an uniform random order. The number of internal lookups is higher for higher levels, although a large fraction of data resides at lower levels. This is because, at higher levels, many internal lookups are negative, as shown in Figure 4(a)(ii). The number of positive internal lookups is as expected: higher in lower levels as shown in Figure 4(a)(iii). This result shows that files at higher levels serve many negative lookups and thus are worth optimizing. While bloom filters may already make these negative lookups faster, the index block still needs to be searched (before the filter query).

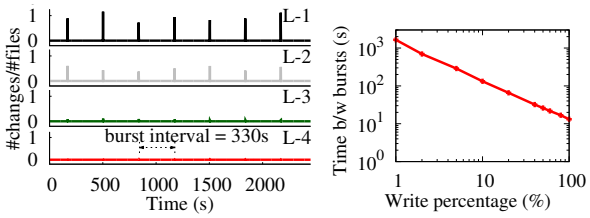
We also conduct the same experiment with another workload where the access pattern follows a zipfian distribution (most requests are to a small set of keys). Most of the results exhibit the same trend as the random workload except for the number of positive internal lookups, as shown in Figure 4(a)(iv). Under the zipfian workload, higher level files also serve numerous positive lookups, because the workload accesses a small set of keys which are often updated and thus stored in higher levels.

Figure 4(b) shows the result when the dataset is sequentially loaded, i.e., keys are inserted in ascending order. In contrast to the randomly-loaded case, there are no negative lookups because keys of different sstable files do not overlap even across levels; the *FindFiles* step finds the one file that may contain the key. Thus, lower levels serve more lookups and can have more benefits from learning. From these observations, we arrive at the next two learning guidelines.

Learning guideline - 3: Do not neglect files at higher levels. Although files at lower levels live longer and serve many lookups, files at higher levels can still serve many negative lookups and in some cases, even many positive lookups. Thus, learning files at higher levels can make both internal lookups faster.

Learning guideline - 4: Be workload- and data-aware. Although most data resides in lower levels, if the workload does not lookup that data, learning those levels will yield less benefit; learning thus must be aware of the workload. Further, the order in which the data is loaded influences which levels receive a large fraction of internal lookups; thus, the system must also be data-aware. The amount of internal lookups acts as a proxy for both the workload and load order. Based on the amount of internal lookups, the system must dynamically decide whether to learn a file or not.

Lifetime of Levels. Given that a level as a whole can also be learned, we now measure and analyze the lifetimes of levels.



(a) Timeline of changes (b) Time between bursts for L4

Figure 5: **Changes at Levels.** *(a) shows the timeline of file creations and deletions at different levels. Note that #changes/#files is higher than 1 in L_1 as there are more creations and deletions than the number of files. (b) shows the time between bursts for L4 for different write percentages.*

Level learning cannot be applied at L_0 because it is unsorted: files in L_0 can have overlapping key ranges. Once a level is learned, any change to the level causes a re-learning. A level changes when new sstables are created at that level, or existing ones are deleted. Thus, intuitively, a level would live for an equal or shorter duration than the individual sstables. However, learning at the granularity of a level has the benefit that the candidate sstables need not be found in a separate step; instead, upon a lookup, the model just outputs the sstable and the offset within it.

We examine the changes to a level by plotting the timeline of file creations and deletions at L_1, L_2, L_3, and L_4 in Figure 5(a) for a 5%-write workload; we do not show L_0 for the reason above. On the y-axis, we plot the number of changes divided by the total files present at that level. A value of 0 means there are no changes to the level; a model learned for the level can be used as long as the value remains 0. A value greater than 0 means that there are changes in the level and thus the model has to re-learned. Higher values denote a larger fraction of files are changed.

First, as expected, we observe that the fraction of files that change reduces as we go down the levels because lower levels hold a large volume of data in many files, confirming our intuition. We also observe that changes to levels arrive in bursts. These bursts are caused by compactions that cause many files at a level to be rewritten. Further, these bursts occur at almost the same time across different levels. The reason behind this is that for the dataset we use, levels L_0 through L_3 are full and thus any compaction at one layer results in cascading compactions which finally settle at the non-full L_4 level. The levels remain static between these bursts. The duration for which the levels remain static is longer with a lower amount of writes; for example, with 5% writes, as shown in the figure, this period is about 5 minutes. However, as the amount of writes increases, the lifetime of a level reduces as shown in Figure 5(b); for instance, with 50% writes, the lifetime of L_4 reduces to about 25 seconds. From these observations, we arrive at our final learning guideline.

Learning guideline - 5: Do not learn levels for write-heavy workloads. Learning a level as a whole might be more appro-

priate when the amount of writes is very low or if the workload is read-only. For write-heavy workloads, level lifetimes are very short and thus will induce frequent re-learnings.

Summary. We analyzed how LSMs behave internally by measuring and analyzing the lifetimes of sstable files and levels, and the amount of lookups served by files at different levels. From our analysis, we derived five learning guidelines. We next describe how we incorporate the learning guidelines in an LSM-based storage system.

4 Bourbon Design

We now describe BOURBON, an LSM-based store that uses learning to make indexing faster. We first describe the model that BOURBON uses to learn the data (§4.1). Then, we discuss how BOURBON supports variable-size values (§4.2) and its basic learning strategy (§4.3). We finally explain BOURBON's cost-benefit analyzer that dynamically makes learning decisions to maximize benefit while reducing cost (§4.4).

4.1 Learning the Data

As we discussed, data can be learned at two granularities: individual sstables or levels. Both these entities are sorted datasets. The goal of a model that tries to learn the data is to predict the location of a key in such a sorted dataset. For example, if the model is constructed for a sstable file, it would predict the file offset given a key. Similarly, a level model would output the target sstable file and the offset within it.

Our requirements for a model is that it must have low overheads during learning and during lookups. Further, we would like the space overheads of the model to be small. We find that piecewise linear regression (PLR) [4, 27] satisfies these requirements well; thus, BOURBON uses PLR to model the data. The intuition behind PLR is to represent a sorted dataset with a number of line segments. PLR constructs a model with an error bound; that is, each data point d is guaranteed to lie within the range $[d_{pos} - \delta, d_{pos} + \delta]$, where d_{pos} is the predicted position of d in the dataset and δ is the error bound specified beforehand.

To train the PLR model, BOURBON uses the Greedy-PLR algorithm [47]. Greedy-PLR processes the data points one at a time; if a data point cannot be added to the current line segment without violating the error bound, then a new line segment is created and the data point is added to it. At the end, Greedy-PLR produces a set of line segments that represents the data. Greedy-PLR runs in linear time with respect to the number of data points.

Once the model is learned, inference is quick: first, the correct line segment that contains the key is found (using binary search); within that line segment, the position of the target key is obtained by multiplying the key with the line's slope and adding the intercept. If the key is not present in the predicted position, a local search is done in the range determined by the error bound. Thus, lookups take $O(log\text{-}s)$ time, where s is the number of segments, in addition to a constant time to do the local search. The space overheads of PLR are small: a few tens of bytes for every line segment.

Other models or algorithms such as RMI [31], PGM-Index [19], or splines [29] may also be suitable for LSMs and may offer more benefits than PLR. We leave their exploration within LSMs for future work.

Workload	Baseline time (s)	File model		Level model	
		Time(s)	% model	Time(s)	% model
Mixed: Write-heavy	82.6	71.5 (1.16 ×)	74.2	95.1 (0.87 ×)	1.5
Mixed: Read-heavy	89.2	62.05 (1.44 ×)	99.8	74.3 (1.2 ×)	21.4
Read-only	48.4	27.2 (1.78 ×)	100	25.2 (1.92 ×)	100

Table 1: **File vs. Level Learning.** *The table compares the time to perform 10M operations in baseline WiscKey, file-learning, and level-learning. The numbers within the parentheses show the improvements over baseline. The table also shows the percentage of lookups that take the model path; remaining take the original path because the models are not rebuilt yet.*

4.2 Supporting Variable-size Values

Learning a model that predicts the offset of a key-value pair is much easier if the key-value pairs are the same size. The model then can multiply the predicted position of a key by the size of the pair to produce the final offset. However, many systems allow keys and values to be of arbitrary sizes.

BOURBON requires keys to be of a fixed size, while values can be of any size. We believe this is a reasonable design choice because most datasets have fixed-size keys (e.g., user-ids are usually 16 bytes), while value sizes vary significantly. Even if keys vary in size, they can be padded to make all keys of the same size. BOURBON supports variable-size values by borrowing the idea of key-value separation from WiscKey [37]. With key-value separation, sstables in BOURBON just contain the keys and the pointer to the values; values are maintained in the value log separately. With this, BOURBON obtains the offset of a required key-value pair by getting the predicted position from the model and multiplying it with the record size (which is *keysize* + *pointersize*.) The value pointer serves as the offset into the value log from which the value is finally read.

4.3 Level vs. File Learning

BOURBON can learn individual sstables files or entire levels. Our analysis in the previous section showed that files live longer than levels under write-heavy workloads, hinting that learning at the file granularity might be the best choice. We now closely examine this tradeoff to design BOURBON's basic learning strategy. To do so, we compare the performance of file learning and level learning for different workloads. We initially load a dataset and build the models. For the read-only workload, the models need not be re-learned. In the mixed workloads, the models are re-learned as data changes. The results are shown in Table 1.

For mixed workloads, level learning performs worse than file learning. For a write-heavy (50%-write) workload, with level learning, only a small percentage of internal lookups are able to use the model because with a steady stream of incoming writes, the system is unable to learn the levels. Only a mere 1.5% of internal lookups take the model path; these lookups are the ones performed just after loading the data and when the initial level models are available. We observe that all the 66 attempted level learnings failed because the level changed before the learning completed. Because of the additional cost of re-learnings, level learning performs even worse than the baseline with 50% writes. On the other hand, with file models, a large fraction of lookups benefit from the models and thus file learning performs better than the baseline. For read-heavy mixed workload (5%), although level learning has benefits over the baseline, it performs worse than file learning for the same reasons above.

Level learning can be beneficial for read-only settings: as shown in the table, level learning provides 10% improvements over file learning. Thus, deployments that have only read-only workloads can benefit from level learning. Given that BOURBON's goal is to provide faster lookups while supporting writes, levels are not an appropriate choice of granularity for learning. Thus, BOURBON uses file learning by default. However, BOURBON supports level learning as a configuration option that can be useful in read-only scenarios.

4.4 Cost vs. Benefit Analyzer

Before learning a file, BOURBON must ensure that the time spent in learning is worthwhile. If a file is short-lived, then the time spent learning that file wastes resources. Such a file will serve few lookups and thus the model would have little benefit. Thus, to decide whether or not to learn a file, BOURBON implements an online cost vs. benefit analysis.

4.4.1 Wait Before Learning

As our analysis showed, even in the lower levels, many files are short-lived. To avoid the cost of learning short-lived files, BOURBON waits for a time threshold, T_{wait}, before learning a file. The exact value of T_{wait} presents a cost vs. performance tradeoff. A very low T_{wait} leads to some short-lived files still being learned, incurring overheads; a large value causes many lookups to take the baseline path (because there is no model built yet), thus missing opportunities to make lookups faster. BOURBON sets the value of T_{wait} to the time it takes to learn a file. Our approach is never more than a factor of two worse than the optimal solution, where the optimal solution knows apriori the lifetime and decides to either immediately or never learn the file (i.e., it is two-competitive [25]). Through measurements, we found that the maximum time to learn a file (which is at most ∼4MB in size) is around 40 ms on our experimental setup. We conservatively set T_{wait} to be 50 ms in BOURBON's implementation.

4.4.2 To Learn a File or Not

BOURBON waits for T_{wait} before learning a file. However, learning a file even if it lives for a long time may not be beneficial. For example, our analysis shows that although lower-level files live longer, for some workloads and datasets, they serve relatively fewer lookups than higher-level files; higher-level files, although short-lived, serve a large percentage of negative internal lookups in some scenarios. BOURBON, thus, must consider the potential benefits that a model can bring, in addition to considering the cost to build the model. It is profitable to learn a file if the benefit of the model (B_{model}) outweighs the cost to build the model (C_{model}).

***Estimating* $\mathbf{C_{model}}$.** One way to estimate C_{model} is to assume that the learning is completely performed in the background and will not affect the rest of the system; i.e., C_{model} is 0. This is true if there are many idle cores which the learning threads can utilize and thus do not interfere with the foreground tasks (e.g., the workload) or other background tasks (e.g., compaction). However, BOURBON takes a conservative approach and assumes that the learning threads will interfere and slow down the other parts of the system. As a result, BOURBON assumes C_{model} to be equal to T_{build}. We define T_{build} as the time to train the PLR model for a file. We find that this time is linearly proportional to the number of data points in the file. We calculate T_{build} for a file by multiplying the average time to a train a data point (measured offline) and the number of data points in the file.

***Estimating* $\mathbf{B_{model}}$.** Estimating the potential benefit of learning a file, B_{model}, is more involved. Intuitively, the benefit offered by the model for an internal lookup is given by $T_b - T_m$, where T_b and T_m are the average times for the lookup in baseline and model paths, respectively. If the file serves N lookups in its lifetime, the net benefit of the model is: $B_{model} = (T_b - T_m) * N$. We divide the internal lookups into negative and positive because most negative lookups terminate at the filter, whereas positive ones do not; thus,

$$B_{model} = ((T_{n.b} - T_{n.m}) * N_n) + ((T_{p.b} - T_{p.m}) * N_p)$$

where N_n and N_p are the number of negative and positive internal lookups, respectively. $T_{n.b}$ and $T_{p.b}$ are the time in the baseline path for a negative and a positive lookup, respectively; $T_{n.m}$ and $T_{p.m}$ are the model counterparts.

B_{model} for a file cannot be calculated without knowing the number of lookups that the file will serve or how much time the lookups will take. The analyzer, to estimate these quantities, maintains statistics of files that have lived their lifetime, i.e., files that were created, served many lookups, and then were replaced. To estimate these quantities for a file F, the analyzer uses the statistics of other files at the same level as F; we consider statistics only at the same level because these statistics vary significantly across levels.

Recall that BOURBON waits before learning a file. During this time, the lookups are served in the baseline path. BOURBON uses the time taken for these lookups to estimate

$T_{n.b}$ and $T_{p.b}$. Next, $T_{n.m}$ and $T_{p.m}$ are estimated as the average negative and positive model lookup times of other files at the same level. Finally, N_n and N_p are estimated as follows. The analyzer first takes the average negative and positive lookups for other files in that level; then, it is scaled by a factor $f = s/\bar{s}_l$, where s if the size of the file and $\bar{s}_l$ is the average file size at this level. While estimating the above quantities, BOURBON filters out very short-lived files.

While bootstrapping, the analyzer might not have enough statistics collected. Therefore, initially, BOURBON runs in an always-learn mode (with T_{wait} still in place.) Once enough statistics are collected, the analyzer performs the cost vs. benefit analysis and chooses to learn a file if $C_{model} < B_{model}$, i.e., benefit of a model outweighs the cost. If multiple files are chosen to be learned at the same time, BOURBON puts them in a max priority queue ordered by $B_{model} - C_{model}$, thus prioritizing files that would deliver the most benefit.

Our cost-benefit analyzer adopts a simple scheme of using average statistics of other files at the same level. While this approach has worked well in our initial prototype, using more sophisticated statistics and considering workload distributions (e.g., to account for keys with different popularity) could be more beneficial. We leave such exploration for future work.

4.5 Bourbon: Putting it All Together

We describe how the different pieces of BOURBON work together. Figure 6 shows the path of lookups in BOURBON. As shown in (a), lookups can either be processed via the model (if the target file is already learned), or in the baseline path (if the model is not built yet.) The baseline path in BOURBON is similar to the one shown in Figure 1 for LevelDB, except that BOURBON stores the values separately and so *ReadValue* reads the value from the log.

Once BOURBON learns a sstable file, lookups to that file will be processed via the learned model as shown in Figure 6(b). ① *FindFiles*: BOURBON finds the candidate sstables; this step required because BOURBON uses file learning. ② *LoadIB+FB*: BOURBON loads the index and filter blocks; these blocks are likely to be already cached. ③ *ModelLookup*: BOURBON performs a look up for the desired key k in the candidate sstable's model. The model outputs a predicted position of k within the file (*pos*) and the error bound (δ). From this, BOURBON calculates the data block that contains records $pos - \delta$ through $pos + \delta$.[†] ④ *SearchFB*: The filter for that block is queried to check if k is present. If present, BOURBON calculates the range of bytes of the block that must be loaded; this is simple because keys and pointers to values are of fixed size. ⑤ *LoadChunk*: The byte range is loaded. ⑥ *LocateKey*: The key is located in the loaded chunk. The key will likely be present in the predicted position (the midpoint of the loaded chunk); if not, BOURBON performs a binary search in the chunk. ⑦ *ReadValue*: The value is read from the value log using the pointer.

[†]Sometimes, records $pos - \delta$ through $pos + \delta$ span multiple data blocks; in such cases, BOURBON consults the index block (which specifies the maximum key in each data block) to find the data block for *pos*.

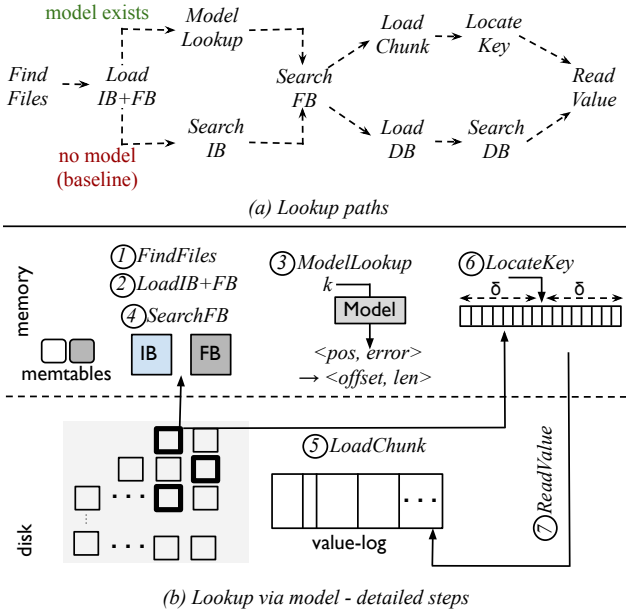


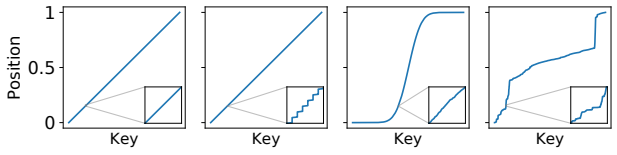
Figure 6: **BOURBON Lookups.** *(a) shows that lookups can take two different paths: when the model is available (shown at the top), and when the model is not learned yet and so lookups take the baseline path (bottom); some steps are common to both paths. (b) shows the detailed steps for a lookup via a model; we show the case where models are built for files.*

Possible improvements. Although BOURBON's implementation is highly-optimized and provides many features common to real systems, it lacks a few features. For example, in the current implementation, we do not support string keys and key compression (although we support value compression). For string keys, one approach we plan to explore is to treat strings as base-64 integers and convert them into 64-bit integers, which could then adopt the same learning approach described herein. While this approach may work well for small keys, large keys may require larger integers (with more than 64 bits) and thus efficient large-integer math is likely essential. Also, BOURBON does not support adaptive switching between level and file models; it is a static configuration. We leave supporting these features to future work.

5 Evaluation

To evaluate BOURBON, we ask the following questions:

- Which portions of lookup does BOURBON optimize? (§5.1)
- How does BOURBON perform with models available and no writes? How does performance change with datasets, load orders, and request distributions? (§5.2)
- How does BOURBON perform with range queries? (§5.3)



(a) Linear (b) Seg10% (c) Normal (d) OSM

Figure 7: **Datasets.** *The figure shows the cumulative distribution functions (CDF) of three synthetic datasets (linear, segmented-10%, and normal) and one real-world dataset (OpenStreetMaps). Each dataset is magnified around the 15% percentile to show a detailed view of its distribution.*

- In the presence of writes, how does BOURBON's cost-benefit analyzer perform compared to other approaches that always or never re-learn? (§5.4)
- Does BOURBON perform well on real benchmarks? (§5.5)
- Is BOURBON beneficial when data is on storage? (§5.6)
- Is BOURBON beneficial with limited memory? (§5.7)
- What are the error and space tradeoffs of BOURBON? (§5.8)

Setup. We run our experiments on a 20-core Intel Xeon CPU E5-2660 machine with 160-GB memory and a 480-GB SATA SSD. We use 16B integer keys and 64B values, and set the error bound of BOURBON's PLR as 8. Unless specified, our workloads perform 10M operations. We use a variety of datasets. We construct four synthetic datasets: linear, segmented-1%, segmented-10% , and normal, each with 64M key-value pairs. In the linear dataset, keys are all consecutive. In the seg-1% dataset, there is a gap after a consecutive segment of 100 keys (i.e., every 1% causes a new segment). The segmented-10% dataset is similar, but there is a gap after 10 consecutive keys. We generate the normal dataset by sampling 64M unique values from the standard normal distribution $N(0,1)$ and scale to integers. We also use two real-world datasets: Amazon reviews (AR) [5] and New York OpenStreetMaps (OSM) [2]. AR and OSM have 33.5M and 21.9M key-value pairs, respectively. These datasets vary widely in how the keys are distributed. Figure 7 shows the distribution for a few datasets. Most of our experiments focus on the case where the data resides in memory; however, we also analyze cases where data is present on storage.

5.1 Which Portions does BOURBON Optimize?

We first analyze which portions of the lookup BOURBON optimizes. We perform 10M random lookups on the AR and OSM datasets and show the latency breakdown in Figure 8. As expected, BOURBON reduces the time spent in indexing. The portion marked *Search* in the figure corresponds to *SearchIB* and *SearchDB* in the baseline, versus *ModelLookup* and *LocateKey* in BOURBON. The steps in BOURBON have lower latency than their baseline counterparts. Interestingly, BOURBON reduces data-access costs too, because BOURBON loads a smaller byte range than the entire block loaded by the baseline.

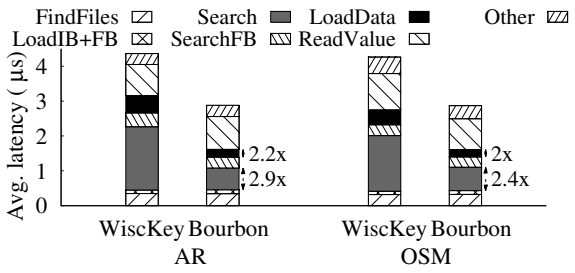
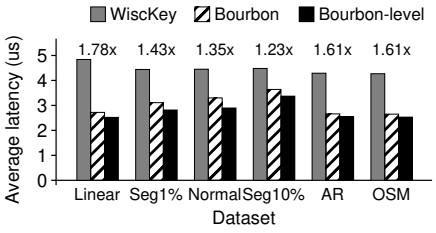


Figure 8: **Latency Breakdown.** *The figure shows latency breakdown for WiscKey and* BOURBON. *Search denotes SearchIB and SearchDB in WiscKey; the same denotes ModelLookup and LocateKey in* BOURBON. *LoadData denotes LoadDB in WiscKey; the same denotes LoadChunk in* BOURBON. *These two steps are optimized by* BOURBON *and are shown in solid colors; the number next to a step shows the factor by which it is made faster in* BOURBON.



(a) Average lookup latency

Dataset	#segs	latency (μs)
Linear	900	2.72
Seg1%	640K	3.11
Normal	705K	3.3
Seg10%	6.4M	3.64
AR	129K	2.66
OSM	295K	2.65

(b) Number of segments

Figure 9: **Datasets.** *(a) compares the average lookup latencies of* BOURBON, BOURBON*-level, and WiscKey for different datasets; the numbers on the top show the improvements of* BOURBON *over WiscKey. (b) shows the number of segments for different datasets in* BOURBON.

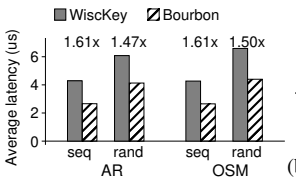
5.2 Performance under No Writes

We next analyze BOURBON's performance when the models are already built and there are no updates. For each experiment, we load a dataset and allow the system to build the models; during the workload, we issue only lookups.

5.2.1 Datasets

To analyze how the performance is influenced by the dataset, we run the workload on all six datasets and compare BOURBON's lookup performance against WiscKey. Figure 9 show the results. As shown in 9(a), BOURBON is faster than WiscKey for all datasets; depending upon the dataset, the improvements vary (1.23× to 1.78×). BOURBON provides the most benefit for the linear dataset because it has the smallest number of segments (one per model); with fewer segments, fewer searches are needed to find the target line segment. From 9(b), we observe that latencies increase with the number of segments (e.g., latency of seg-1% is greater than that of linear). We cannot compare the number of segments in AR and OSM with others because the size of these datasets is significantly different.

Level learning. Given that level learning is suitable for read-only scenarios, we configure BOURBON to use level learn-



(a) Average latency

Dataset	Positive		Negative	
	#	Speedup	#	Speedup
AR	10M	2.15×	23M	1.83×
OSM	10M	1.99×	22M	1.82×

(b) Positive vs. negative internal lookups for randomly loaded case

Figure 10: **Load Orders.** *(a) shows the performance for AR and OSM datasets for sequential (seq) and random (rand) load orders. (b) compares the speedup of positive and negative internal lookups.*

ing and analyze its performance. As shown in Figure 9(a), BOURBON-level is 1.33× – 1.92× faster than the baseline. BOURBON-level offers more benefits than BOURBON because a level-model lookup is faster than finding the candidate sstables and then doing a file-model lookup. This confirms that BOURBON-level is an attractive option for read-only scenarios. However, since level models only provide benefits for read-only workloads and give at most 10% improvement compared to file models, we focus on BOURBON with file learning for our remaining experiments.

5.2.2 Load Orders

We now explore how the order in which the data is loaded affects performance. For this experiment, we use the AR and OSM datasets and load them in two ways: sequential (keys are inserted in ascending order) and random (keys are inserted in an uniformly random order). With sequential loading, sstables do not have overlapping key ranges even across levels; whereas, with random loading, sstables at one level can overlap with sstables at other levels.

Figure 10 shows the result. First, regardless of the load order, BOURBON offers significant benefit over baseline (1.47× – 1.61×). Second, the average lookup latencies increase in the randomly-loaded case compared to the sequential case (e.g., 6μs vs. 4μs in WiscKey for the AR dataset). This is because while there are no negative internal lookups in the sequential case, there are many (23M) negative lookups in the random case (as shown in 10(b)). Thus, with random load, the total number of internal lookups increases by 3×, increasing lookup latencies.

Next, we note that the speedup over baseline in the random case is less than that of the sequential case (e.g., 1.47× vs. 1.61× for AR). Although BOURBON optimizes both positive and negative internal lookups, the gain for negative lookups is smaller (as shown in 10(b)). This is because most negative lookups in the baseline and BOURBON end just after the filter is queried (filter indicates absence); the data block is not loaded or searched. Given there are more negative than positive lookups, BOURBON offers less speedup than the sequential case. However, this speedup is still significant (1.47×).

5.2.3 Request Distributions

Next, we analyze how request distributions affect BOURBON's performance. We measure the lookup latencies under six request distributions: sequential, zipfian, hotspot, exponential, uniform, and latest. We first randomly load the AR and OSM datasets and then run the workloads; thus, the data can be segmented and there can be many negative internal lookups. As shown in Figure 11, BOURBON makes lookups faster by 1.54× – 1.76× than the baseline. Overall, BOURBON reduces latencies regardless of request distributions.

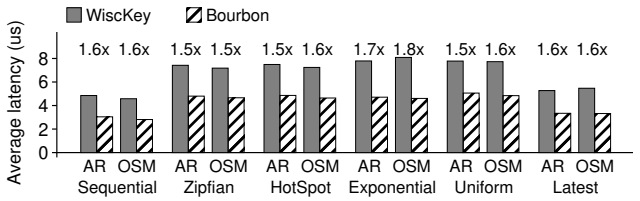


Figure 11: **Request Distributions.** *The figure shows the average lookup latencies of different request distributions from AR and OSM datasets.*

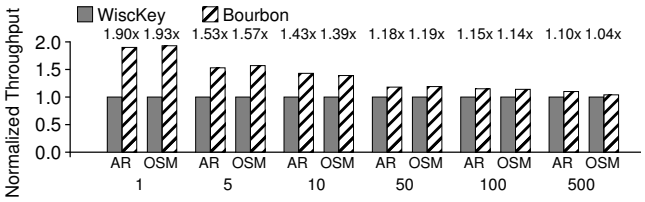


Figure 12: **Range Queries.** *The figure shows the normalized throughput of range queries with different range lengths from AR and OSM datasets.*

Read-only performance summary. When the models are already built and when there are no writes, BOURBON provides significant speedup over baseline for a variety of datasets, load orders, and request distributions.

5.3 Range Queries

We next analyze how BOURBON performs on range queries. We perform 1M range queries on the AR and OSM datasets with various range lengths. Figure 12 shows the throughput of BOURBON normalized to that of WiscKey. With short ranges, where the indexing cost (i.e., the cost to locate the first key of the range) is dominant, BOURBON offers the most benefit. For example, with a range length of 1 on the AR dataset, BOURBON is 1.90× faster than WiscKey. The gains drop as the range length increases; for example, BOURBON is only 1.15× faster with queries that return 100 items. This is because, while BOURBON can accelerate the indexing portion, it follows a similar path as WiscKey to scan subsequent keys. Thus, with large range lengths, indexing accounts for less of the total performance, resulting in lower gains.

5.4 Efficacy of Cost-benefit Analyzer with Writes

We next analyze how BOURBON performs in the presence of writes. Writes modify the data and so the models must be re-learned. In such cases, the efficacy of BOURBON's cost-benefit analyzer (cba) is critical. We thus compare

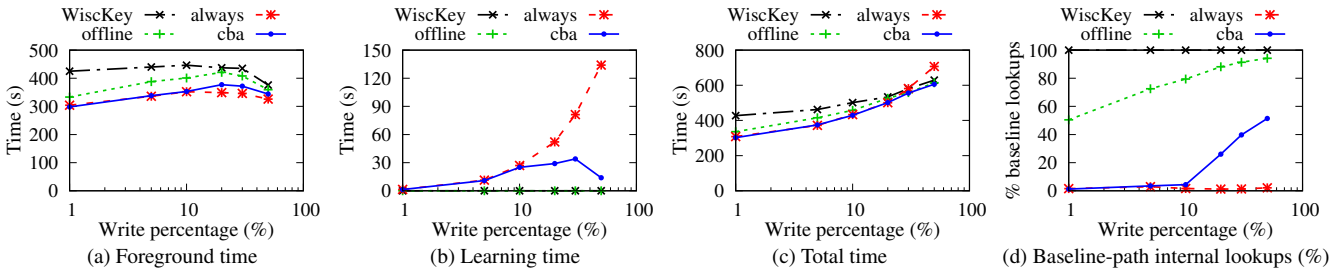


Figure 13: **Mixed Workloads.** *(a) compares the foreground times of WiscKey,* BOURBON*-offline (offline),* BOURBON*-always (always), and* BOURBON*-cba (cba); (b) and (c) compare the learning time and total time, respectively; (d) shows the fraction of internal lookups that take the baseline path.*

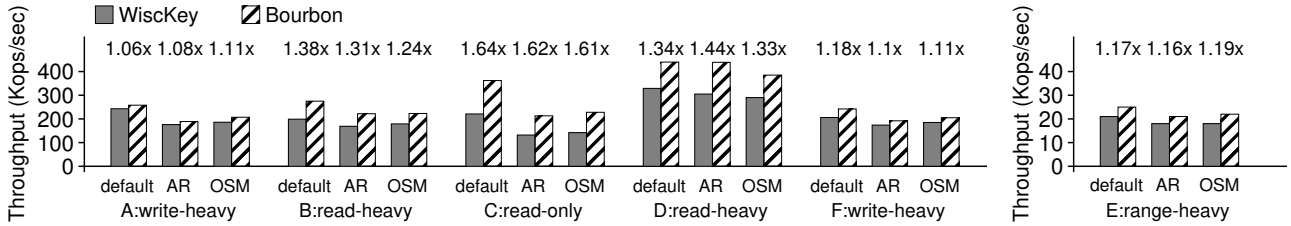


Figure 14: **Macrobenchmark-YCSB.** *The figure compares the throughput of* BOURBON *against WiscKey for six YCSB workloads across three datasets.*

BOURBON's cba against two strategies: BOURBON-offline and BOURBON-always. BOURBON-offline performs no learning as writes happen; models exist only for the initially loaded data. BOURBON-always re-learns the data as writes happen; it always decides to learn a file without considering cost. BOURBON-cba re-learns as well, but it uses the cost-benefit analysis to decide whether or not to learn a file.

We run a workload that issues 50M operations with varying percentages of writes on the AR dataset. To calculate the total amount of work performed for each workload, we sum together the time spent on the foreground lookups and inserts (Figure 13(a)), the time spent learning (13(b)), and the time spent on compaction (not shown); the total amount of work is shown in Figure 13(c). The figure also shows the fraction of internal lookups that take the baseline path (13(d)).

First, as shown in 13(a), all BOURBON variants reduce the workload time compared to WiscKey. The gains are lower with more writes because BOURBON has fewer lookups to optimize. Next, BOURBON-offline performs worse than BOURBON-always and BOURBON-cba. Even with just 1% writes, a significant fraction of internal lookups take the baseline path in BOURBON-offline as shown in 13(d); this shows re-learning as data changes is crucial.

BOURBON-always learns aggressively and thus almost no lookups take the baseline path even for 50% writes. As a result, BOURBON-always has the lowest foreground time. However, this comes at the cost of increased learning time; for example, with 50% writes, BOURBON-always spends about 134 seconds learning. Thus, the total time spent increases with more writes for BOURBON-always and is even higher than baseline WiscKey as shown in 13(c). Thus, aggressively learning is not ideal.

Given a low percentage of writes, BOURBON-cba decides to learn almost all the files, and thus matches the characteristics of BOURBON-always: both have a similar fraction of lookups taking the baseline path, both require the same time learning, and both perform the same amount of work. With a high percentage of writes, BOURBON-cba chooses not to learn many files, reducing learning time; for example, with 50% writes, BOURBON-cba spends only 13.9 seconds in learning (10× lower than BOURBON-always). Consequently, many lookups take the baseline path. BOURBON-cba takes this action because there is less benefit to learning as the data is changing rapidly and there are fewer lookups. Thus, it almost matches the foreground time of BOURBON-always. But, by avoiding learning, the total work done by BOURBON-cba is significantly lower.

Summary. Aggressive learning offers fast lookups but with high costs; no re-learning provides little speedup. Neither is ideal. In contrast, BOURBON provides high benefits similar to aggressive learning while lowering total cost significantly.

5.5 Real Macrobenchmarks

We next analyze how BOURBON performs under two real benchmarks: YCSB [10] and SOSD [28].

5.5.1 YCSB

We use six workloads that have different read-write ratios and access patterns: A (w:50%, r:50%), B (w:5%, r:95%), C (read-only), D (read latest, w:5%, r:95%), E (range-heavy, w:5%, range:95%), F (read-modify-write:50%, r:50%). We use three datasets: YCSB's default dataset (created using

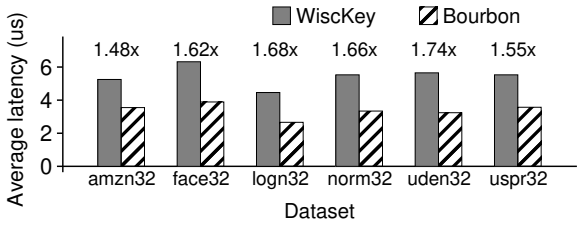


Figure 15: **Macrobenchmark-SOSD.** *The figure compares lookup latencies from the SOSD benchmark. The numbers on the top show* BOURBON*'s improvements over the baseline.*

Dataset	WiscKey latency (μs)	BOURBON	
		Latency(μs)	Speedup
Amazon Reviews (AR)	3.53	2.75	1.28×
NewYork OpenStreetMaps (OSM)	3.14	2.51	1.25×

Table 2: **Performance on Fast Storage.** *The table shows* BOURBON*'s lookup latencies when the data is stored on an Optane SSD.*

ycsb-load [3]), AR, and OSM, and load them in a random order. Figure 14 shows the results.

For the read-only workload (YCSB-C), all operations benefit and BOURBON offers the most gains (about 1.6×). For read-heavy workloads (YCSB-B and D), most operations benefit, while writes are not improved and thus BOURBON is 1.24× – 1.44× faster than the baseline. For write-heavy workloads (YCSB-A and F), BOURBON improves performance only a little (1.06× – 1.18×). First, a large fraction of operations are writes; second, the number of the internal lookups taking the model path decreases (by about 30% compared to the read-heavy workload because BOURBON chooses not to learn some files). YCSB-E consists of range queries (range lengths varying from 1 to 100) and 5% writes. BOURBON reaches 1.16× – 1.19× gain. In summary, as expected, BOURBON improves the performance of read operations; at the same time, BOURBON does not affect the performance of writes.

5.5.2 SOSD

We next measure BOURBON's performance on the SOSD benchmark designed for learned indexes [28]. We use the following six datasets: book sale popularity (amzn32), Facebook user ids (face32), lognormally (logn32) and normally (norm32) distributed datasets, uniformly distributed dense (uden32) and sparse (uspr32) integers. Figure 15 shows the average lookup latency. As shown, BOURBON is about 1.48× – 1.74× faster than the baseline for all datasets.

5.6 Performance on Fast Storage

Our analyses so far focused on the case where the data resides in memory. We now analyze if BOURBON will offer benefit when the data resides on a fast storage device. We run a read-only workload on sequentially loaded AR and OSM datasets on an Intel Optane SSD. Table 2 shows the result. Even when the data is present on a storage device, BOURBON offers benefit (1.25× – 1.28× faster lookups than WiscKey).

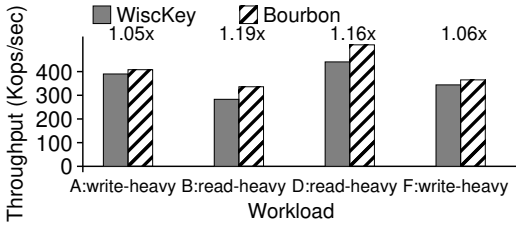


Figure 16: **Mixed Workloads on Fast Storage.** *The figure compares the throughput of* BOURBON *against WiscKey for four read-write mixed YCSB workloads. We use the YCSB default dataset for this experiment.*

Workload	WiscKey latency (μs)	BOURBON	
		Latency(μs)	Speedup
Uniform	98.6	94.4	1.04×
Zipfian	18.8	15.1	1.25×

Table 3: **Performance with Limited Memory.** *The table shows* BOURBON*'s average lookup latencies from the AR dataset on a machine with a SATA SSD and limited memory.*

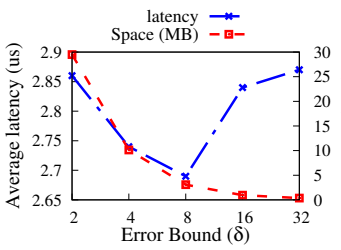
Figure 16 shows the result for read-write mixed YCSB workloads on the same device with the default YCSB datasest. As expected, while BOURBON's benefits are marginal for write-heavy workloads (YCSB-A and YCSB-F), it offers considerable speedup (1.16× – 1.19×) for read-heavy workloads (YCSB-B and YCSB-D). With the emerging storage technologies (e.g., 3D XPoint memory), BOURBON will offer even more benefits.

5.7 Performance with Limited Memory

We further show that, even with no fast storage and limited available memory, BOURBON can still offer benefit with skewed workloads, such as zipfian. We experiment on a machine with a SATA SSD and memory that only holds about 25% of the database. We run a uniform random workload, and a zipfian workload with consecutive hotspots where 80% of the requests access about 25% of the database. Table 3 shows the result. With the uniform workload, BOURBON is only 1.04× faster because most of the time is spent loading the data into the memory. With the zipfian workload, in contrast, indexing time instead of data-access time dominates because a large number of requests access the small portion of data that is already cached in memory. BOURBON is able to reduce this significant indexing time and thus offers 1.25× lower latencies.

5.8 Error Bound and Space Overheads

We finally discuss the characteristics of BOURBON's ML model, specifically its error bound (δ) and space overheads. Figure 17(a) plots the error bound (δ) against the average lookup latency (left y-axis) for AR dataset. As δ increases, fewer line segments are created, leading to fewer searches, thus reducing latency. However, beyond $\delta = 8$, although the time to find the segment reduces, the time to search within a segment increases, thus increasing latency. We find that BOURBON's choice of $\delta = 8$ is optimal for other datasets too.



(a) Error-bound tradeoff

Dataset	Space Overheads	
	MB	%
Linear	0.02	0.0
Seg1%	15.38	0.21
Seg10%	153.6	2.05
Normal	16.94	0.23
AR	3.09	0.08
OSM	7.08	0.26

(b) Space overheads

Figure 17: **Error-bound Tradeoffs and Space Overheads.** *(a) shows how the PLR error bound affects lookup latency and memory overheads; (b) shows the space overheads for different datasets.*

Figure 17(a) also shows how space overheads (right y-axis) vary with δ. As δ increases, fewer line segments are created, leading to low space overheads. Table 17(b) shows the space overheads for different datasets. As shown, for a variety of datasets, the overhead compared to the total dataset size is little (0% – 2%).

6 Related Work

Learned indexes. The core idea of our work, replacing indexing structures with ML models, is inspired from the pioneering work on learned indexes [31]. However, learned indexes do not support updates, an essential operation that an storage-system index must support. Recent research tries to address this limitation. For instance, XIndex [45], FITing-Tree [20], and AIDEL [35] support writes using an additional array (delta index) and with periodic re-training, whereas Alex [13] uses gapped array at the leaf nodes of a B-tree to support writes.

Most prior efforts optimize B- tree variants, while our work is the first to deeply focus on LSMs. Further, while most prior efforts implement learned indexes to stand-alone data structures, our work is the first to show how learning can be integrated and implemented into an existing, optimized, production-quality system. While SageDB [30] is a full database system that uses learned components, it is built from scratch with learning in mind. Our work, in contrast, shows how learning can be integrated into an existing, practical system. Finally, instead of "fixing" new read-optimized learned index structures to handle writes (like previous work), we incorporate learning into an already write-optimized, production-quality LSM.

LSM optimizations. Prior work has built many LSM optimizations. Monkey [11] carefully adjusts the bloom filter allocations for better filter hit rates and memory utilization. Dostoevsky [12], HyperLevelDB [16], and bLSM [42] develop optimized compaction policies to achieve lower write amplification and latency. cLSM [23] and RocksDB [18] use non-blocking synchronization to increase parallelism. We take a different yet complimentary approach to LSM optimization by incorporating models as auxiliary index structures to improve lookup latency, but each of the others are orthogonal and compatible to our core design.

Model choices. Duvignau et al. [14] compare a variety of piecewise linear regression algorithms. Greedy-PLR, which we utilize, is a good choice to realize fast lookups, low learning time, and small memory overheads. Neural networks are also widely used to approximate data distributions, especially datasets with complex non-linear structures [34]. However, theoretical analysis [36] and experiments [43] show that training a complex neural network can be prohibitively expensive. Similar to Greedy-PLR, recent work proposes a one-pass learning algorithm based on splines [29] and identifies that such an algorithm could be useful for learning sorted data in LSMs; we leave their exploration within LSMs for future work.

7 Conclusion

In this paper, we examine if learned indexes are suitable for write-optimized log-structured merge (LSM) trees. Through in-depth measurements and analysis, we derive a set of guidelines to integrate learned indexes into LSMs. Using these guidelines, we design and build BOURBON, a learned-index implementation for a highly-optimized LSM system. We experimentally demonstrate that BOURBON offers significantly faster lookups for a range of workloads and datasets.

BOURBON is an initial work on integrating learned indexes into an LSM-based storage system. More detailed studies, such as more sophisticated cost-benefit analysis, general string support, and different model choices, could be promising for future work. In addition, we believe that BOURBON's learning approach may work well in other write-optimized data structures such as the B^{ϵ}-tree [6] and could be an interesting avenue for future work. While our work takes initial steps towards integrating learning into production-quality systems, more studies and experience are needed to understand the true utility of learning approaches.

Acknowledgements

We thank Alexandra Fedorova (our shepherd) and the anonymous reviewers of OSDI '20 for their insightful comments and suggestions. We thank the members of ADSL for their excellent feedback. We also thank CloudLab [41] for providing a great environment to run our experiments and reproduce our results during artifact evaluation. This material was supported by funding from NSF grants CNS-1421033, CNS-1763810 and CNS-1838733, Intel, Microsoft, Seagate, and VMware. Aishwarya Ganesan is supported by a Facebook fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] BadgerDB. https://github.com/dgraph-io/badger.

[2] Open Street Maps. https://www.openstreetmap.org/#map=4/38.01/-95.84.

[3] Running a Workload. https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload.

[4] Jayadev Acharya, Ilias Diakonikolas, Jerry Li, and Ludwig Schmidt. Fast Algorithms for Segmented Regression. *arXiv preprint arXiv:1607.03990*, 2016.

[5] Amazon. Amazon Customer Reviews Dataset. https://registry.opendata.aws/amazon-reviews/.

[6] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to b^ε-trees and write-optimization. *;login: Operating Systems and Sysadmin*, (5):23–28, Oct 2015.

[7] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, and Jiakai Zhang. End to End Learning for Self-driving Cars. *arXiv preprint arXiv:1604.07316*, 2016.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, WA, November 2006.

[9] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), June 1979.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IA, June 2010.

[11] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD '17)*, Chicago, IL, May 2017.

[12] Niv Dayan and Stratos Idreos. Dostoevsky: Better Space-time Trade-offs for LSM-tree based Key-value Stores via Adaptive removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.

[13] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David Lomet. ALEX: An Updatable Adaptive Learned Index. *arXiv preprint arXiv:1905.08898*, 2019.

[14] Romaric Duvignau, Vincenzo Gulisano, Marina Papatriantafilou, and Vladimir Savic. Piecewise linear approximation in data streaming: Algorithmic implementations and experimental analysis. *arXiv preprint arXiv:1808.08877*, 2018.

[15] Sarah M Erfani, Sutharshan Rajasegarar, Shanika Karunasekera, and Christopher Leckie. High-dimensional and Large-scale Anomaly Detection using a Linear One-class SVM with Deep Learning. *Pattern Recognition*, 58:121–134, 2016.

[16] Robert Escriva, Sanjay Ghemawat, David Grogan, Jeremy Fitzhardinge, and Chris Mumford. HyperLevelDB. https://github.com/rescrv/HyperLevelDB, 2013.

[17] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A Guide to Deep Learning in Healthcare. *Nature medicine*, 25(1):24–29, 2019.

[18] Facebook. RocksDB. http://rocksdb.org/.

[19] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index. *Proceedings of the VLDB Endowment*, 13(10):11621175, Jun 2020.

[20] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*, Amsterdam, Netherlands, June 2019.

[21] Lars George. *HBase: The Definitive Guide: Random Access to Your Planet-size Data*. OReilly Media, Inc., 2011.

[22] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[23] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data

stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–14, 2015.

[24] Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[25] Anna R Karlin, Kai Li, Mark S Manasse, and Susan Owicki. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. *ACM SIGOPS Operating Systems Review*, 25(5):41–55, 1991.

[26] Andrej Karpathy. Software 2.0. https://medium.com/@karpathy/software-2-0-a64152b37c35, November 2017.

[27] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An Online Algorithm for Segmenting Time Series. In *Proceedings 2001 IEEE international conference on data mining*, 2001.

[28] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. SOSD: A Benchmark for Learned Indexes, 2019.

[29] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A Single-Pass Learned Index. *arXiv preprint arXiv:2004.14541*, may 2020.

[30] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A Learned Database System. In *Proceedings of 9th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2019.

[31] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*, Houston, TX, June 2018.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, Lake Tahoe, NV, December 2012.

[33] Avinash Lakshman and Prashant Malik. Cassandra – A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky Resort, Montana, Oct 2009.

[34] Stéphane Lathuilière, Pablo Mesejo, Xavier Alameda-Pineda, and Radu Horaud. A comprehensive analysis of deep regression. *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[35] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. A Scalable Learned Index Scheme in Storage Systems. *arXiv preprint arXiv:1905.06256*, 2019.

[36] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. In *Advances in neural information processing systems*, pages 855–863, 2014.

[37] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[38] Mathias Meyer. The Riak Handbook, 2012.

[39] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[40] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shangai, China, October 2017.

[41] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[42] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, AZ, May 2012.

[43] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104. IEEE, 2016.

[44] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the

Game of Go With Deep Neural Networks and Tree Search. 529(7587), January 2016.

[45] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: A Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 308–320, 2020.

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. https://arxiv.org/abs/1609.08144, September 2016.

[47] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. Maximum Error-bounded Piecewise Linear Representation for Online Stream Approximation. *The VLDB journal*, 23(6), 2014.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network

Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim†,
Henry Hoffmann, and Haryadi S. Gunawi

University of Chicago †Surya University

Abstract

This paper presents LinnOS, an operating system that leverages a light neural network for inferring SSD performance at a very fine—per-IO—granularity and helps parallel storage applications achieve performance predictability. LinnOS supports black-box devices and real production traces without requiring any extra input from users, while outperforming industrial mechanisms and other approaches. Our evaluation shows that, compared to hedging and heuristic-based methods, LinnOS improves the average I/O latencies by 9.6-79.6% with 87-97% inference accuracy and 4-6µs inference overhead for each I/O, demonstrating that it is possible to incorporate machine learning inside operating systems for real-time decision-making.

1 Introduction

Predictable performance is an important requirement for today's and future systems [19, 51, 55, 65]. For data-center systems serving web search, email, and many other types of interactive services, predictable latency is even more important. On the bright side, faster and faster SSDs are available and becoming a dominant factor in the storage market [10]. On the negative side, SSD internal complexity continues to grow, and achieving highly predictable latency on modern flash devices remains an open challenging problem.

Due to the intrinsic NAND idiosyncrasies, modern flash devices behave like an "operating system," managing all of its internal resources with background operations such as garbage collection, buffer flushing, wear leveling, and read repairs. While important, these are the kinds of operations that pose a threat to latency predictability [15, 25, 27, 30, 49, 53, 71, 76], which is still a fresh problem faced by many storage industries in recent years [4, 31, 50, 57]. Furthermore, with a report that flash devices contribute to more than 19% of the total response time for some online applications [76], more solutions should be explored.

Because the device itself cannot mask the unpredictable latency, a vast amount of research has been devoted to this space. "White-box" approaches—that re-architect device internals [17, 33, 34, 36, 47, 61, 68, 71]—are powerful, but face a high barrier to adoption unless SSD vendors implement the recommendations. In the middle ground, "gray-box" methods suggest partial device-level modification combined with OS or application-level changes working together in taming the latency unpredictability [38, 39, 40, 58, 76, 77]. However, they also depend on the vendors' willingness to modify the device interface. Finally, more adoptable "black-box" techniques attempt to mask the unpredictability without modifying the underlying hardware and its level of abstraction. Some of them optimize the file systems or storage applications specifically for SSD usage [18, 37, 41, 42, 43, 54, 59, 69, 70], while some others simply use speculative execution [1, 5] but pay the cost of extra I/Os due to being oblivious to storage behaviors. Among all the approaches above, arguably, the most popular solution is speculative execution given its simplicity and capability to mitigate *every* slow I/O. For example, "hedged requests" [21], a form of speculative execution, is supported in many widely-used key-value stores today [1, 5, 8].

We take a new approach: let the device be the device (black-box) and do not redesign the file systems or applications, but *learn* the device behavior (*i.e.*, not be storage oblivious). The key to our approach is learning. Can we learn the behavior of the underlying device in a black-box way and use the results of the learning to increase predictability, so applications can know in advance whether their performance expectations can be fulfilled? This is a domain that machine learning can likely help. We introduce LinnOS, an operating system that has the capability of learning and inferring *per-I/O* speed with high accuracy and minimal overhead using a lightweight neural network. We show how LinnOS helps storage applications, in particular storage arrays/clusters with built-in failover logic (*e.g.*, flash RAID, Cassandra, MongoDB), achieve extreme latency predictability on unpredictable flash storage.

The biggest challenge for LinnOS is to be as effective and fine-grained as the popular approach, speculative execution, which can mitigate *every* slow I/O by sending a duplicate I/O to another node or device. Speculative execution's success in increasing predictability comes at the cost of poor resource utilization. The key to avoiding this cost is to know the current activities going on inside the devices and always schedule I/Os to those devices that will provide faster responses.

However, because keeping the abstraction barrier is a fundamental constraint, we need to learn to infer latency and make the inference highly usable. Achieving this requires *learning and inferring on a very fine, per-I/O scale* in a live fashion. To the best of our knowledge, there is no existing learning approach for I/O scheduling that supports such fine-grained learning due to the challenges of achieving per-I/O accuracy and fast online inference. To address this, LinnOS introduces three technical contributions.

First, LinnOS converts the hard latency inference problem into a simple *binary* inference ("fast" or "slow" speed). We take advantage of the typical latency distributions in system deployments, specifically, a behavior that forms a Pareto distribution with a high alpha number. In other words, most of the time (*e.g.*, >90%), the latency is very stable, but occasionally (*e.g.*, <10% of the time), the latency exhibits a long-tail behavior [16, 21, 45, 53]. The behavior of flash storage reflects the same distribution [15, 26]. In this simple view where users only want "slow" I/Os to become "fast," inferring the exact latencies is overkill. With this intuition, LinnOS comes with an algorithm that monitors the latency distribution of the current workload running on the flash device and computes a roughly optimal threshold that separates the slow and fast speed ranges.

Second, with the binary model, LinnOS employs a simple admission control for clustered storage applications. LinnOS makes a binary inference on every incoming I/O using a light neural network model that infers the I/O speed in advance in a black-box manner without any guide from the device nor application. If the I/O is inferred to be fast, LinnOS will submit it to the flash device; otherwise it will revoke the I/O and inform the application. With this timely and straightforward binary information, the storage application can quickly failover the I/O to another node or device that holds the same replica. Furthermore, resources are efficiently utilized because the original slow I/O has been revoked.

Third, LinnOS balances the accuracy and performance of the neural network. High accuracy but high inference time will lead to a significant per-I/O overhead, especially for modern SSDs. On the other hand, lowering inference time by lowering accuracy will lead to many false inferences that make storage performance hard to reason about.

For high accuracy, LinnOS profiles the latency of millions of I/Os submitted to the device (a natural "data lake"), which will be used to train the neural network. Furthermore, as we convert regression to a simple binary classification, the output accuracy is significantly improved (akin to the simplicity of "cat or dog" image classification). The next challenge is to decide the input features that matter most to improving accuracy. We will present our surprising findings. For example, "important-looking" features such as block offsets, read/write flags, or long history of writes do not play a significant role. In the end, the input features become tractable with only two types of information: the latencies of a few recently completed I/Os and the number of pending I/Os when those I/Os and the current, to-be-inferred I/O arrived.

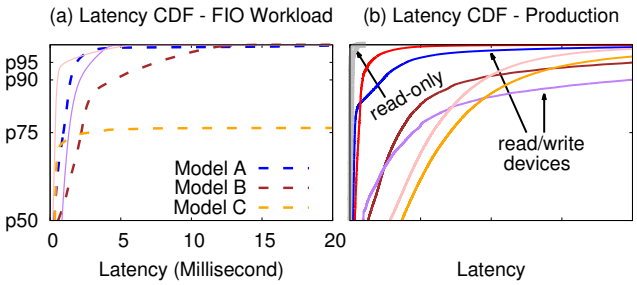


Figure 1: **Latency distribution.** *The figures show CDFs of block-level read latencies, as discussed in Section 2. For the left figure, we ran one FIO workload on five different SSD models (the five CDF lines). For the right figure, we plot the latencies of seven block-level traces obtained from four read-write servers (colored lines) and two read-only servers (bold gray lines) in Azure, Bing, and Cosmos clusters. The x-axis is anonymized for proprietary reasons. The traces are available from Microsoft with NDA.*

For performance, the challenge is to make an inference (admission decision) in sub-10µs, which is crucial as we target fine-grained live inference for fast storage devices. While using deeper models with more features can improve accuracy, it will hurt inference latency and would be too expensive for usage in the I/O layer. Through several design iterations, we cut the inference time to 4-6µs with minor accuracy loss, achieved with several methods: a 3-layer light neural network, weight quantization, and (optional) 2-threaded/2-core matrix multiplication.

Our evaluation shows that LinnOS supports a wide variety of black-box devices (10 device models tested) and works on real production traces without requiring any extra input from users (*e.g.*, hints about traces/devices or latency deadlines etc.), outperforms industrial approaches such as pure hedging, and beats simple and "advanced" heuristics that we design. Compared to these methods, LinnOS, complemented by hedging based on the learning outcome, further improves the average I/O latencies by 9.6-79.6% with 87-97% accuracy and only 4-6µs inference overhead for every I/O.

Overall, we show that it is plausible to adopt machine learning methods for operating systems to learn black-box devices. We conclude with many interesting discussions to explore in the future. LinnOS code is made public.

2 Background

Unpredictability. To motivate the problem, the colored lines in Figure 1a show read latency distribution in a read-write workload running on five different SSD models ranging from consumer SATA and NVMe SSDs to new data-center ones. Model A delivers fast and stable latencies up to about "p98" (the 98^{th} percentile), but models B and C exhibit larger la-

tency tails starting at p90 and p75, respectively. However, when the write operations are converted into read I/Os, the performance becomes highly stable without much latency tail (not shown in the figure). Figure 1b also confirms this in real production scenarios in Microsoft SSD-backed servers. The colored lines show block-level read latencies of read-write servers (more variability), and the gray lines for read-only servers (more predictability). All of these confirm how write-triggered garbage collection (GC), buffer flushing, and other internal operations are contending with user read I/Os. We only address read performance unpredictability because we found write latencies to be (surprisingly) stable as they are absorbed by the internal memory buffer on the device, hence not affected by internal contentions. Write latency spikes only happen when the buffer is full (rarely happened due to internal periodic flush).

Internal complexities. Inferring when a flash drive is exhibiting tail latency is hard given the internal complexities that factor into latency behavior. As a couple of examples, I/Os contend with each other if they fall into the same chip or channel, which depends on the hidden striping and partitioning logic; two user I/Os that go to separate channels might have different fates when one channel is occupied by GC data transfers between the chips in the channel. Our internal findings show that SSDs can have wide layouts (*e.g.*, 32 channels with four chips per channel) or deep layouts (*e.g.*, four channels with 16 chips per channel), where the latter will cause more channel contention. Some SSDs employ large write buffers from 256MB to as small as 12 MB and can periodically flush from every 3ms to as high as one second. As shown in Figure 1, this internal contention can affect from 1% to 25% of all read requests.

In this context, modern storage applications usually apply a "wait-then-speculate" approach that is agnostic about the device's internal complexities. For example, with hedging, applications wait for a timeout (*e.g.*, the p95 latency), then issue extra speculative I/Os, and use whichever is the faster response. Speculative execution works well for coarse-grained tasks (tens to hundreds of seconds), but is ineffective for flash storage since the waiting is costly when the expected response time is less than a few milliseconds (§5.3).

Machine learning. Before we tried machine learning techniques such as neural networks, we asked whether simple heuristics would be accurate enough in inferring per-I/O speed. For example, one might assume that a long I/O queue length implies longer latencies—a heuristic that works well for spinning disks [13, 62, 64]. However, for SSDs, due to the internal complexities, queue length is not highly correlated with delay (we did not find a high Pearson's correlation or Spearman's correlation between queue length and I/O latency). We also created a more "advanced" heuristic, but it did not yield a satisfying result (more in the evaluation section). While it is possible to keep crafting the right heuristic that can adapt to different workloads and device models, we decided to resort to machine learning. Recent operating and distributed systems research successfully employed machine learning for resource allocation and scheduling [22, 23, 24, 28, 48, 51, 52, 60]. A similar exploration targeting the I/O layer can lead to a powerful result, as we show in this paper.

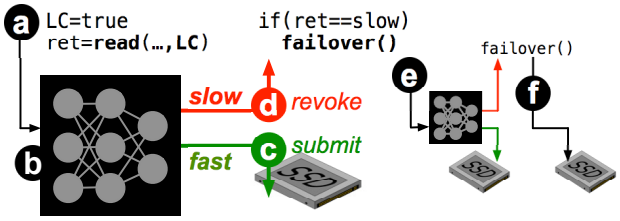


Figure 2: **Usage scenario.** *This usage scenario is explained in Section 3.1. "LC" implies latency critical.*

3 Overview

We now give the overview of LinnOS, its usage scenario, architecture, and challenges, followed by its design (§4).

3.1 Usage Scenario

LinnOS is beneficial for parallel, redundant storage such as flash arrays (cluster-based or RAID) that maintain multiple replicas of the same block, as illustrated in Figure 2. **(a)** With LinnOS, when a storage application performs an I/O via OS system calls, it can add a one-bit flag, hinting to LinnOS that the I/O is latency-critical (`LC=true`), *e.g.*, for interactive services. Such tagging of critical operations has been proposed many times [73, 76], but in our case, the bit is used to trigger LinnOS to infer the I/O latency. **(b)** Before submitting the I/O to the underlying SSD, LinnOS inputs the I/O information to the neural network model that it has trained, which will make a binary inference: fast or slow. **(c)** If the output is "fast," LinnOS *submits* the I/O down to the device. **(d)** Otherwise, if it is "slow," LinnOS *revokes* the I/O (not entered to the device queue) and returns a "`slow`" error code. **(e)** Upon receiving the error code, the storage application can failover the same I/O to another replica. **(f)** In the worst case where the application must failover to the last replica, this last retry will not be tagged as latency-critical so that the I/O will complete and not be revoked.

3.2 Overall Architecture

Figure 3 shows LinnOS's overall architecture, which consists of five main components.

(a) The model. At the center of LinnOS is the speedy inference model (Section 4.3) with a light neural network. The model's input features are information about the current outstanding I/Os and recently completed ones. The model infers

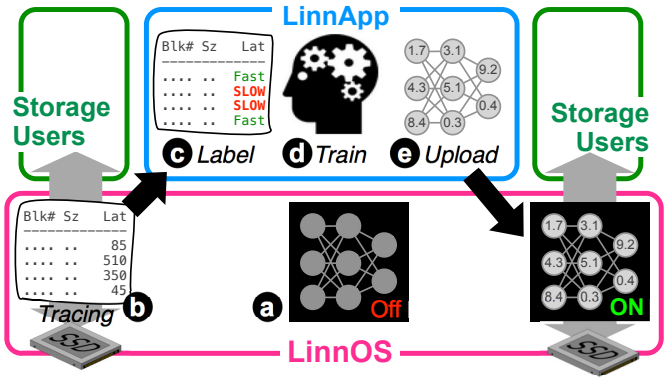


Figure 3: **LinnOS architecture.** *The figure displays LinnOS architecture including LinnApp, as summarized in Section 3.2. The two SSD pictures represent the same SSD instance; the left one depicts tracing/training and the right one live inference on the SSD.*

the speed of every incoming I/O individually. The model's output is the binary inference about the I/O (fast/slow).

(b) Tracing. To train the model, LinnOS uses the current live workload that the SSD is serving. To have a rich representative dataset, this can be done during normal busy hours. The I/O metadata (block offset, size, read/write) and their resulting latencies are recorded using `blktrace`. With millions of I/Os collected, this naturally forms the "data lake" of our model. The training data (the collected trace) is expected to be different than the "test data" (the I/Os that will be inferred when the model is activated).

(c) Labeling with inflection point analysis. The collected trace is then supplied to LinnApp, a supporting user-level application. LinnApp has three main jobs: labeling, training, and uploading trained weights to LinnOS. Because the model is designed to produce a binary output, the model must be trained with two labels, "fast" and "slow." Hence, given a latency distribution in the trace, LinnApp runs an algorithm (§4.2.1) that finds the "inflection point," a latency value that divides the fast and slow latency ranges.

(d) Training. With this inflection point, LinnApp labels the traced I/Os with "fast" and "slow" labels and proceeds with the training phase (using TensorFlow). We emphasize the labeling is done automatically *without* human input. This training phase can be run anywhere, on GPU or CPU nodes.

(e) Uploading weights. The training phase generates the weights for the neurons in the model that will be uploaded to LinnOS. Because using floating points is not well supported in OS kernel, the weights are converted to integers by quantization. The model is then activated, and LinnOS is ready to make inferences and revoke "slow" I/Os.

3.3 Challenges

Using a machine learning approach for making online, fine-grained inferences on I/O speed requires us to solve the following fundamental challenges.

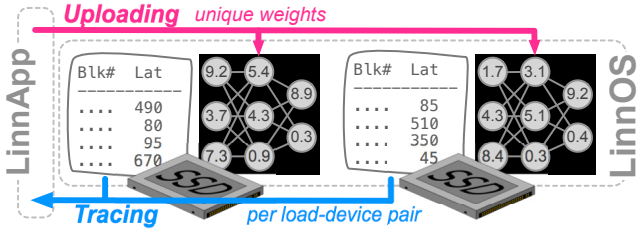


Figure 4: **Anticipating heterogeneity.** *The figure shows heterogeneous trained models, as mentioned in Section 3.3.*

High accuracy. The inference must be accurate. We should not revoke I/Os that can be served fast ("false revoke") or submit those that will be slow ("false submit"). Accuracy depends on careful output labeling and input features selection. If the label classification is too complicated, high accuracy is hard to achieve, *e.g.*, we find that classification by linear bucketing (0-10, 10-20µs, etc.) or exponential bucketing (0-1, 2-4µs, etc.) is hard to make accurate and should remain as a future work. However, the simple two-class approach (fast or slow) simplifies the output into a binary format, which helps the model achieve high accuracy.

Fast inference. For modern SSDs, while the raw NAND read latency is advertised to be below 100µs, we see that for typical production workload on data-center SSDs (Section 5), the actual user-perceived latency is above 200µs more than 50% of the time. Given this observation, we believe the challenge is to do decision-making in around 5µs, a $<3\%$ overhead per I/O. Fast inference depends on input preprocessing, the depth of the layers, neuron complexity, and feature representation. Using deep layers that tend to improve accuracy is not attractive in our problem domain. The input features must be minimized to include only the features that matter. Hence, we must balance accuracy and performance. Moreover, considering that operating systems run on CPUs, the models must be CPU-friendly [67].

Anticipating heterogeneity. In flash arrays (RAID or cluster-based), the user load is not always balanced, and all the flash hardware might not be homogeneous. Because this heterogeneity can lead to different latency distributions observed on different devices, we should not use one global latency value (*e.g.*, 1ms inflection point) to differentiate fast and slow speed for all the devices. For example, 3ms perhaps could be considered fast enough on slower SSDs or the ones with heavier user load. While we do not expect that the heterogeneity will be extreme (*e.g.*, a good storage system typically balances the load very well), heterogeneity is still important to address. For this reason, LinnApp collects per-device traces and trains the model for *every* load-device pair in the array (Figure 4). After the training phase completes, LinnApp supplies the model weights to all instances of LinnOS in a cluster-based array or to one instance of LinnOS in a RAID-based array. In the latter, LinnOS carries N trained

models for the N drives in the RAID. Furthermore, to anticipate workload changes over time, LinnApp occasionally recollects traces (*e.g.*, every few hours) to check if the inflection point has shifted significantly such that the model must be retrained.

4 LinnOS Design

In this section, we describe our solution to the challenges mentioned above. To the best of our knowledge, LinnOS is the first operating system that successfully infers I/O speed in a *fast*, *accurate*, *live*, *fine-grained*, and *general* fashion. The key to this is the "lightness" of the neural network model that LinnOS employs. This section presents the final design and the principal intuitions about how we get there. We will explain LinnOS design chronologically, from data collection (§4.1), labeling via inflection point analysis (§4.2), the model design (§4.3), and how to improve its accuracy (§4.4) and performance (§4.5), and summarize its advantages (§4.6).

4.1 Training Data Collection

This project started with a simple question: can we infer the performance of every I/O accurately? Since we use machine learning, accuracy depends on the amount of true-signal data available, the more, the better. Fortunately, I/O systems inherently can collect a large amount of data. Given low-overhead tracing tools and hundreds of KIOPS of workload that modern SSDs can serve, collecting a large amount of data for training is not an issue (a large "I/O data lake").

For every load-SSD pair to model, LinnApp collects traces of the real workload running on the drive. For example, for inferring a production workload performance on a particular SSD in deployment, an online trace will be collected. For every I/O, we collect five raw fields, the submission time, block offset, block size, read/write, and most importantly, the I/O completion time. Because the model input (Section 4.3) does not necessarily take the same raw fields, in this phase, we also convert the fields to the input feature format.

The main challenge here is to decide how long the trace should be. If the behavior of the training data (the latency distribution) is very different from that of the "test" data (the to-be-inferred I/Os), the inference accuracy will drop. In this work, we take a simple approach where we use a busy-hour trace (*e.g.*, midday). In the evaluation (§5.2), we show that for production workloads, a busy-hour trace well represents the other hours, *i.e.*, the inflection point does not deviate much. As mentioned above, to anticipate a dramatic shift in workload behavior, retracing and retraining can be done.

4.2 Labeling (with Inflection Point)

As we employ a supervised classification approach, the model must be trained with labels. If we label every I/O with the actual µs-level latency, there will be too many labels for our problem domain; a user might not care if the I/O is delayed by 1µs. Another option is to use a linear (0-10µs, 10-20µs, and so on) or exponential labeling (2-4µs, 4-8µs, and so on). While these fit better, the model is still hard to make accurate and fast after many design iterations. The accuracy only reached 60-70% because many times, an I/O that should fall into a specific group (*e.g.*, 128-256µs) is often mis-inferred to the neighbor groups (*e.g.*, 256-512µs)—"a Lhasa Apso dog can easily be misidentified as a Shih Tzu dog." This is perhaps why prior successes in auto-learning storage performance were only done at a coarse-grained level such as average latency or throughput aggregated for many requests [29, 66, 74].

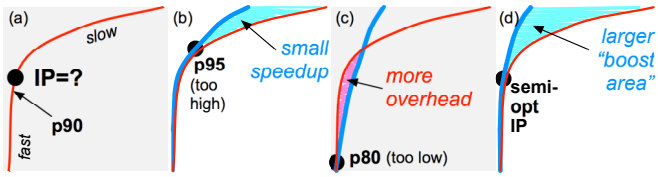


Figure 5: **Inflection point (fast/slow threshold).** *The figures show the results of using a higher, lower, or semi-optimum inflection point (IP) for the fast/slow threshold as explained in Section 4.2. The figure format is latency CDF, as in Figure 1.*

With all this mind and an understanding of how performance variance behaves in the field [15, 21, 26, 45, 49], we observe that latencies often form a Pareto distribution with a high alpha number [7]. As an example shown in Figure 5a, 90% of the time, the latency is likely stable, but in the other 10% of the time, it starts forming a long tail. Such a Pareto distribution clearly contrasts the fast and slow regions. Hence, a simple conjecture can be made that users only worry about the tail behavior, not the precise latency.

To separate the two regions, we need to find the "best" *inflection point* (marked with "**IP**=?" in Figure 5a) for maximizing the latency reduction. Setting the inflection point too relaxed (*e.g.*, the p95 latency in Figure 5b) will make LinnOS treat the relatively slow I/Os between p90 and p95 as "fast" (no failover), reducing the scope for effective retries, hence failing to cut many tail latencies, as highlighted by the small shaded area between the original and projected distributions (more in §4.2.1) in Figure 5b. On the other hand, setting the inflection point too low (*e.g.*, the p80 latency in Figure 5c) will make LinnOS revoke too many I/Os, including those that are supposed to be fast, which will induce unnecessary retry overhead as shown in Figure 5c.

An optimum inflection point implies that for every slow I/O that will be revoked, it is likely that the other replicas can serve it fast within the same time frame. Likewise, for every fast I/O, it should not be failed over. Finding this optimum point will deliver the maximum gap between the original tail-heavy and tail-free distributions, as shown by the large shaded area in Figure 5d. Finding an optimum value how-

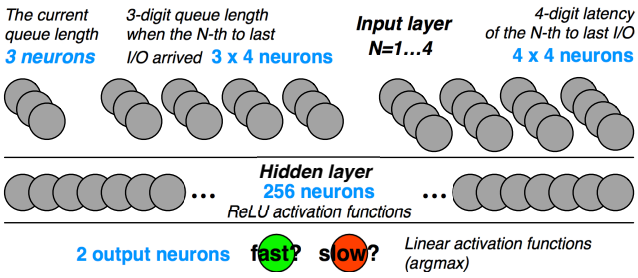


Figure 6: **Light neural network.** *The figure depicts LinnOS 3-layer neural network explained in Section 4.3.*

ever is hard in practice, fundamentally because of the many unknowns: we do not know which replica the request will be failed over to (application dependent); the training data is only an approximation of the future unknown test data; other variability such as CPU or network contention can factor into unknown retry overhead. The next section describes our best-effort algorithm in finding a semi-optimum inflection point for every workload-device pair.

4.2.1 Inflection Point Algorithm

First, during data collection, we collect t workload traces (T_1 to T_t) running on d devices (D_1 to D_d), respectively, where $t{=}{=}d$. Every trace T_i gives us the latency distribution of the workload running on the device (as in Figure 5a). To find the unique inflection point (IP) value for every $T_i{-}D_i$ pair, we run a user-space simulation based on random replica selection, with the assumption that latency delay is independent across the SSDs. For illustrative purposes, we use specific device numbers (*e.g.*, D_1) in our explanation below.

(1) For every $T_i{-}D_i$ pair, we pick a starting IP value where the slope of the CDF is one (likely entering the tail area). For example, if for D_1, T_1's 45-degree slope is at $y{=}p90.5$ and $x{=}1ms$, then the IP value is initially set to 1ms. **(2)** For the currently simulated device, D_1, we run a simulation of one million I/Os, ($r_{i=0..1000000}$) where each I/O request r_i takes a random latency value from T_1's real latency distribution. We then simulate LinnOS admission control: if the chosen latency is smaller than 1ms (the current IP), the r_i's new latency is set to be the same; else, if it is larger than 1ms, it will be revoked and failed over to another randomly selected node (*e.g.*, D_4) where a new random latency is picked from its trace, T_4, and the admission control is repeated (submit or revoke). We assume three replicas (configurable), hence a request can only be revoked a maximum of two times. **(3)** The simulation produces the new, optimized latencies for all the r_i in workload trace T_1 that previously went to only one device, D_1, but now can be redirected as if LinnOS admission control is activated. These optimized r_i latencies form the new CDF (as in the bold blue line in Figure 5d). Using the original and new CDFs, we can calculate the *area difference* (the shaded "boost area" in Figure 5d), which represents the latency gain if $1ms$ ($p90.5$) is used as the IP value. **(4)** Still, for D_1, we repeat all the steps above by moving $^+/_-0.1$ percentile within the $^+/_-10$ percentile ranges from the initial IP value. For every new IP value, the simulation gives a new boost area. We now can pick the IP^{max}, the IP value that gives us the *largest (positive) boost area*, which will be used as the fast-slow threshold in training the model for device D_1. **(5)** We repeat all the steps for other devices (D_2, D_3, etc.). At the end, for every $T_i{-}D_i$ pair, our algorithm generates a unique IP_i^{max} value. All these steps are repeated upon recalibration (§4.4).

4.3 Light Neural Network Model

Before we decided to build a light neural network model, we explored various learning methods such as logistic regression, decision trees, and random forests. We found that the accuracy only ranges from 17-84%, while a basic neural network can reach a better accuracy. Although it is possible to continue optimizing each of these methods to its full potential, we decided to start from an acceptable baseline that our initial neural model delivered. Below, we describe our final model (Figure 6), from input features, their representation, to the neural layers. We will emphasize how we use storage intuitions to design the model, as opposed to brute-force.

Input features. To infer the speed of every I/O, our model takes three inputs: **(a)** the number of pending I/Os when an incoming I/O arrives (in the number of 4KB pages, including the incoming I/O), **(b)** the latency of the R most-recently completed I/Os, where we set R as 4, and **(c)** the number of pending I/Os at the time when each of the R completed I/Os arrived. We now reason about these necessary inputs.

Deciding the first feature is straightforward—an I/O latency typically correlates with how many I/Os are currently pending. The unit we use here is the number of 4KB pending pages, and the reason is that the lowest granularity of striping inside SSDs is typically at the page level and the main contention is at channel and chip level.

While for disks, the first feature might be sufficient for inferring single-spindle performance, for SSDs, the other two features are required. In essence, to speculate whether the SSD is currently busy internally, we need to record a small piece of historical information, the latencies of the last four I/Os, as well as how many pending I/Os existed when those I/Os arrived. Put simply, if recent I/Os experienced a long delay without many pending I/Os, then the model could learn that there is likely an internal contention due to device-level activities such as GC, internal flushing, or wear leveling. In this case, the model will suggest revoking incoming I/Os until the number of pending I/Os drops substantially so that the device can provide fast responses despite heavy internal activity. Once the device resumes serving I/Os, the model can tell whether the device-level contention is over from the returned latency values.

Our features above look simple because we have removed

unnecessary features after many design iterations. For example, we surprisingly found that important-looking features such as block offsets, read/write flags, or long history of writes do not significantly improve accuracy. We make several conjectures. First, on read/write flags, although NAND-level read/write latencies differ, almost all medium/high-end SSDs employ write buffering. Thus, the problem of read-behind-write is no longer observable. More likely observable is read-behind-buffer-flush delays, which can be learned from our input features. Second, on block offsets, because we target production workloads and the fact that SSDs typically stripe incoming I/Os uniformly across all channels and chips (or with some bounded partitioning), the workload is likely to be evenly scattered, hence block offsets do not really matter for learning. In other words, scenarios where a batch of incoming I/Os with block offsets that simultaneously hit only one chip rarely happen in the field. Third, on history of writes, internal activities such as GC and buffer flush often happen in a short burst, hence they can be sensed by just observing the speed of the last four I/Os. These are surprising but fortunate findings because using just a small set of features will reduce the model's overhead.

Input format. The next challenge is to choose the right input format to be fed to the neurons. First, for the R value, if accuracy is the only important metric, we should record more completed I/Os (the higher R, the better), but it would prolong inference time as the number of neurons would increase. We found that R=4 suffices for balancing performance and accuracy.

In another simplification, we format the number of pending I/Os into three decimal digits. For example, the format for 15 pending I/Os is three integers {0,1,5}. Three digits suffice as device queue length of over 1,000 is rarely heard of. Similarly, for the latencies of the recent completed I/Os, we break the µs latency value into four digits. For example, a latency of a recent I/O that completed in 240µs will be formatted as four features {0,2,4,0}. Latencies larger than 9,999µs will be capped to {9,9,9,9}. In total, our model takes 31 input features, each a one-digit decimal number.

Reformatting the original integers into decimal digits is an effective trade-off. If we use bits and supply every bit to every neuron, there will be too many neurons that increase the model size and hurt inference time. On the other extreme, if every neuron takes a raw integer value, the neurons need to learn over a wide input range, which makes learning/training harder (*e.g.*, latency value can range from 1µs to over 9,999µs). With decimal digits, we make the neuron learning bounded within a small range of 0 to 9.

The network. The final model is a fully-connected neural network with only three layers ("light"), including one input/preprocess layer, one hidden layer, and one output layer, as shown in Figure 6. All the neurons are regular linear neurons (y=wx+b).

The input layer is supplied with the 31 features described above. The raw information from the block layer is converted to the feature format, in an offline way for training and an online way for live inference. For the latter, with some programming optimization, we can achieve O(1) preprocessing overhead. Next, the hidden layer consists of 256 regular neurons. This layer uses RELU activation functions for its low computation cost and ability to support non-linear modeling. More neurons will cause longer inference time and fewer neurons less accuracy. Lastly, the output layer has two neurons with linear activation functions. We use an `argmax` operator to convert the output to a binary decision (*e.g.*, {0.4,0.6} to {0,1}). Overall, this design makes the network lightweight and easy to integrate into the OS, while balancing inference accuracy and performance.

Preceding design iterations. Here we briefly describe how we reach the current design. We started by using the I/O offsets in binary format (32-bit) as the input features since the device FTL mapping basically uses I/O offsets to decide where the I/Os go, which defines the resource contention. This setting allows the learning models to achieve higher accuracies (up to 99% for some traces), however it has a heavy model and high inference overhead, which is impractical for real-time usage. We further trimmed the heavy model but could not find a reasonable tradeoff between generality and inference overhead. As a result, we took a step back from the fine-grained features and switched to more aggregate ones, and finally reached the current design.

4.4 Improving Accuracy

To further improve the model accuracy, we perform false-submit reduction via biased training, model recalibration via retracing/retraining, and inaccuracy masking with high-percentile hedging.

Reducing false submits. An accurate inference means LinnOS submits I/Os that will be fast (true negative) and revokes those that will be slow (true positive). Reversely, inaccurate cases can be categorized into **(a)** "false submit" (false negative) wherein the model believes the request will be served fast, making LinnOS submit the request to the device, but the request will take longer than the fast-slow threshold, or **(b)** "false revoke" (false positive) where the I/O is revoked, but in fact, it can be served fast by the device.

Using the same system intuition on typical latency distributions in the field (Section 4.2), we found that *reducing false submits* is far more important, while false revokes are more tolerable. When the storage devices of a cluster exhibit similar tail behavior (high-alpha Pareto), the *probability* that peer devices are simultaneously busy is relatively *small*. For example, with three replicas and P% busyness, the probability that all the replicas are busy around the same time is $(P/100)^3$ (*e.g.*, 0.000125 with 5% busyness). Another factor is that, with faster networks, a failover cost can be as low

as 1-6µs for flash arrays across PCIe or Fiber Channel or 5-40µs across Ethernet [3] (plus some negligible software overhead).

To summarize, the wrong inference penalty is small for false revokes but high for false submits. In the latter, the I/O will be "stuck" in the device and cannot be revoked. This motivates us to use *biased training* for reducing false submits by allowing more false revokes. We do this by customizing the categorical hinge loss function with a multiplier that puts more penalty weights for false submits, which makes the trained models favor false revokes.

Recalibrating. Another source of inaccuracy happens when the inflection point computed over the training data does not represent the same threshold of the "test" data (the workload during live inference). This can happen under significant workload changes that cause shifts in the latency distributions of the nodes in the cluster. Fortunately, our evaluation of production traces shows that latency distributions do not widely shift across hours (§5.2). However, to anticipate this scenario, re-tracing and re-computation of inflection point analysis can be done periodically every few hours. If in the new workload-device pair, the inflection point has shifted by five percentiles, LinnApp will retrain the model using the newly collected trace and re-upload the new trained weights to the device. Running `blktrace` during the busiest hour in the production workloads we use only generates 300 MB of data (85 KB/s of trace writes) and increases CPU overhead by 0.5% (only relevant parameters are traced).

Masking small inaccuracy. Our methods above managed to increase accuracy up to 98%. Just like other neural networks, achieving 100% accuracy is fundamentally hard and usually implies a lack of generality. Within the small inaccuracy, the long latency tail due to false submits still needs to be circumvented. This is where we marry learning and hedging [21]. When the false submit rate[1] (Section 5.4) is significant (*e.g.*, >5%), we use the rate as an indicator for the hedging percentile value. For example, if 6% of the inferences produce false submits, then p94 hedging will be applied. When the false submit rate is lower, we round up to conventional p95 hedging. Though sometimes this design issues extra I/Os, we show that it can further improve the performance (§5.3).

4.5 Improving Inference Time

A large part of deep neural network (DNN) research mainly focuses on how to structure even larger networks to achieve the highest possible accuracy [11]. Strict latency is often not a constraint. However, putting a neural network into the storage layer poses a unique challenge. Our goal is to reach around 5µs of inference time (as discussed in §3.3), and although the 3-layer design is fundamental to reach the goal, we made further optimizations.

Quantization. First, neuron weights are by default in floating points for improving accuracy, but it is an overkill for our purpose. Some of the major storage functionalities that define contention are striping and partitioning using `mod` operations over integers, which does not require ultra-high precision. Besides, floating point calculations are expensive and hard to manage inside the OS. Hence, we adopt DNN quantization by maintaining precision of three decimal points; the trained floating-point weights are converted to integers with precision of three decimal points. DNN quantization is a popular technique to reduce the space, power, and computation cost of DNN on mobile-platform and IoT devices, albeit some loss on accuracy [20, 32, 72]. In our case, the accuracy loss from quantization is less than 0.1%.

Co-processors. Second, using additional accelerators such as GPUs and TPUs may be possible in the future, but currently, they are optimized towards throughput and do not easily interact with host kernel code. If we move the inference to GPUs, the cross-communication would add more overhead. Furthermore, technology trends suggest that 100-200x improvement on inference latency can be foreseen in the near future with more advanced hardware [6]. This may make LinnOS faster in the future, especially as storage devices are also getting faster. However, until this technology arrives, we show that LinnOS can opportunistically use co-processors (if available) to reduce the average inference time from 6 to 4µs with 2-threaded optimized matrix multiplication using one additional CPU core.

4.6 Summary of Advantages

With all of the techniques, LinnOS delivers advantages in various dimensions, which we show in the evaluation.

- *Performance predictability*. The most important advantage is that LinnOS helps storage applications achieve predictable performance on flash arrays, outperforming other popular methods.
- *Automation*. LinnOS infers I/O operation latency by learning from millions of I/Os and automatically trains and produces neuron weights for different workloads and devices. Storage developers do not have to tweak and configure heuristics manually.
- *Generality*. To achieve predictability, LinnOS does not require device-level modification nor a heavy redesign of file systems or applications. Storage applications simply need to tag latency-critical I/Os. Failover/retry logic is already standard in many storage applications with data replicas.
- *Timeliness*. With fast inference, the application can failover as soon as the `slow` error code is returned, without the need to wait for a timeout.

[1] To clarify, different from conventional way of calculating false positive/negative, in this paper, the false submit rate is based on the submit decision and the actual resulting latency.

- *Efficiency*. With auto-revocation, LinnOS eliminates duplicate I/Os suffered in hedging. Some production systems do not use hedging for the same reason and instead use a more efficient method such as "tied requests," where clones are sent but when one of them is served, the duplicate is canceled [21]. Similar to this "clone-then-cancel" method, our "revoke-then-failover" also avoids duplicates. Furthermore, while some implementation of tied requests burdens the application layer [21], LinnOS supports I/O revocation inside the kernel.
- *Simplicity*. We do not require applications to supply an SLO value such as a deadline [14, 25, 63, 75]. I/O system calls today do not accept SLO info, arguably because setting the proper SLO is not easy [35, 46]. LinnOS simplifies this with an auto-tuned fast/slow binary classification.

4.7 Implementation Complexity

LinnOS extends Linux v5.4.8 in 2170 LOC within the block layer, mostly for the neural network model (written in C) and the simple revocation mechanism. The memory space needed for one neural network model (in total 8706 weights and biases) is 68 KB of kernel memory. LinnApp is written in 3820 LOC including data collection, analysis, labeling, training (using TensorFlow), and quantization. We make the source code public (Section A).

5 Evaluation

In this section, we first describe our evaluation setup (Section 5.1) and then present the results that answer the following important questions:

- *Stability* (Section 5.2): Is our inflection point algorithm stable enough for production workloads?
- *Latency predictability* (Section 5.3): Does LinnOS successfully deliver more predictable latencies compared to other methods?
- *Model accuracy* (Section 5.4): How accurate is the LinnOS neural network in inferring per-I/O speed?
- *Trade-offs* (Section 5.5): What are the performance and accuracy trade-offs in LinnOS?
- *Others* (Section 5.6): How does LinnOS work on other public traces? Can LinnOS support full-stack storage applications? What is the CPU overhead?

5.1 Setup

We present the evaluation workloads, devices, experiments, and methods to which we compare.

Workloads. Our ultimate goal is to evaluate whether LinnOS can help real production scenarios. We use SSD-level traces from Microsoft Azure (**AZ**), Bing Index (**BingI/BI**), Bing Select (**BingS/BS**), and Cosmos (**CO**) servers. Each server type contains I/O traces for six devices. The average trace contains 36 hours of I/O operations.[2] For training data, from each of the four server types, we pick the three busiest device traces and then pick the busiest hour (same three hours); we limit to three due to the number of (expensive) enterprise SSDs that we have (more below). For the "test data" that is dedicated for live experiments, we pick a random time slice from other busy hours, hence training and test data do *not* overlap. Overall, the training and test data do not occupy the entire available traces.

SSD devices. For performance evaluation, we show how much LinnOS helps flash arrays deliver predictable latencies. We prepared two flash arrays with consumer (**"C"**) and enterprise (**"E"**) configurations. The former connects an array of three homogeneous SM951 consumer-level SSDs, and the latter forms three *heterogeneous* enterprise-level SSDs, Intel P4600, Samsung PM1725a, and WD Ultrastar DC SN200. We assume every block is replicated three times across the devices, a typical setup for consumer-facing storage servers. For both configurations, the machine has a 2.6GHz 18-core (36-thread) Intel i9-7980XE CPU with 128GB DRAM. Unless otherwise stated, we do not use accelerators (§4.5). The overhead for failing over revoked I/Os is 15µs. For accuracy evaluation, beyond these four flash models, we also use Intel SSDSC1BG40, Intel SSDSC2BX01, Intel P3700, Intel P4510, Intel S3700, and Samsung 960 EVO, for a total of 10 models. Prior to this evaluation, all devices have been used for months with many workloads that reach the devices' full capacities, hence mimicking devices in the field.

The experiments. For performance evaluation, the experiments are performed with a storage application that executes the traces on the flash arrays, where all the devices serve read/write workloads. For example, in one experiment, the application simultaneously executes three different Azure traces on three separate SM951 devices in the consumer flash array and records the latencies of completed read I/Os. The application has a failover capability to complete revoked I/Os at other devices (as shown earlier in Figure 2). All read I/Os are marked as latency-critical.[3] We are also aware that the traces were collected on medium-end devices at Microsoft (in 2016). Hence, for our high-end flash array configuration, we have to mimic a heavier workload by re-rating the traces to be more intensive. Our methodology is that for each re-rated trace, the resulting baseline latency distribution (after running it on the high-end device) should be similar to the latency distribution in the original trace.

Table 1 shows the I/O characteristics of the re-rated traces. Typically, running these re-rated traces on our drives shows a low slack (<5% of all I/Os), where SSDs see no pend-

[2] The traces are available from Microsoft to the academic community with NDA.

[3] We assume that no write I/Os are latency critical as they are usually absorbed by write buffers. However, if needed, our techniques can be easily integrated with kernels/applications that support write re-routing (*e.g.*, AutoRAID, RAID+).

Test Trace	Avg IOPS	Max IOPS	R:W Ratio	Avg I/O Size Read/Write	Max I/O Size Read/Write
AZ/C	745	4.9K	27:73	24K/18K	64K/64K
BI/C	361	1.8K	17:83	57K/30K	64K/1M
BS/C	114	1.1K	22:78	163K/73K	2M/9M
CO/C	113	623	32:68	479K/121K	6M/32M
AZ/E	13K	31K	25:75	25K/17K	64K/64K
BI/E	2.4K	9.2K	23:77	55K/30K	512K/1M
BS/E	1.3K	4.3K	27:73	196K/73K	2M/9M
CO/E	2.5K	7.2K	22:78	430K/107K	7M/32M

Table 1: **I/O characteristics of re-rated traces (§5.1).** *The upper part (first four rows) is for the consumer-level flash array and the lower is for the enterprise-level one. Every max-IOPS value is measured within a 10-second window.*

ing I/Os for a few milliseconds, and noticeable burstiness (5-30%), where I/Os need to wait in the OS as the SSD queues are full. We believe this accurately emulates the slack and tail behaviors seen in real deployments. Also, the workload bursts across devices are highly correlated, which, in some cases, can cause inevitable long-tail behaviors that no failover can handle. However, in real runs we find that the internal busyness of the devices is not necessarily correlated due to device-level complexities, as LinnOS shows great improvement by evading the underlying device idiosyncrasies (§5.3). All the experiments are repeated three times, and no significant variance was observed.

Methods compared. We perform an extensive evaluation that compares eight methods: baseline, cloning, constant-percentile hedging (*e.g.*, at p95 latency), inflection-point hedging (with our algorithm), simple heuristic, advanced heuristic, LinnOS (by itself), and LinnOS with high-percentile hedging. Comparing LinnOS with white-box approaches [25, 30, 44, 56] is out of the scope of the paper because LinnOS targets black-box devices and we do not have access to an array of programmable devices.

5.2 Inflection Point (IP) Stability

One of the contributions in this paper is finding the semi-optimal fast/slow inflection point (IP) that brings a balance between timeliness and overhead (Figure 5 in Section 4.2). Table 2 shows the IP values our algorithm computed for every workload-device pair. The three numbers in every cell represent three different traces (from the same server type), each running on one of the SSDs in the flash array. As shown, the IP values widely range from p72 to p98, which highlights why a constant timeout value is *not* optimal and hurts performance. These IP values will be used for fast/slow labeling and training, which then generates a unique set of weights for each device.

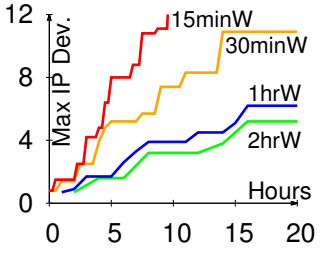


Figure 7: **IP stability.**

	Consumer	Enterprise
Azure	p73.3, p77.0, p91.4	p91.0, p93.2, p97.8
BingIndex	p80.0, p94.5, p98.5	p80.1, p83.3, p97.0
BingSelect	p72.0, p76.9, p87.2	p75.3, p83.7, p86.8
Cosmos	p73.4, p82.5, p84.1	p83.2, p84.8, p95.1

Table 2: **Inflection point (IP) settings.** *This table, as explained in Section 5.2, shows the IP values that our algorithm in Section 4.2.1 computed for every workload-device pair.*

We chose a busy hour (T=`1hr` Window) to collect the training data and calculate the IP values in that time slice. Figure 7 shows the stability of our methodology by plotting the max IP deviations in percentile (y-axis) within the next 20 hours (x-axis) for various T window values. For example, if the chosen hour exhibits p85 IP, but a subsequent hour exhibits p75 or p95 IP, then the deviation is 10 percentiles (y=10). The graph shows that if T=`1hr`Window, the deviation is bounded within five percentiles in the next 15 hours, indicating that frequent retraining is unnecessary. If T is shorter (*e.g.*, `15min`Window), the deviation is more apparent (needs frequent retraining, which typically converges within 15-20 minutes on CPUs, due to LinnOS's light model). If T is larger (`2hr`Window), the gain is not significant. For generality, the figure is the result of our algorithm simulation on all the datasets (36 hours per trace, 24 traces, four server types).

The cost of delayed retraining depends on the deviation. Let us take an example of a model trained for p95 (5ms), but then the workload deviates such that the real IP is at p90 (10ms) because the workload becomes more write-intense. In this case, LinnOS (still using 5ms) will over-revoke many IOs that could have finished before 10ms (more false revokes). If the failover overhead is negligible, this will not cause much harm. Another scenario is when the workload deviates such that the IP moves up to p99 (3ms). Here, LinnOS would over-accept (more false submits) because 3-5ms latency is inaccurately considered "fast," but actually can be made faster. This is where LinnOS without retraining hurts.

5.3 Latency Predictability

We now evaluate LinnOS's success in achieving extreme latency predictability. Figure 8 shows the average I/O latencies (user-perceived) on the two flash arrays (consumer and enterprise) across the eight methods. In more detail, Figure 9 shows the latencies at specific percentiles (p80 to p99.99 in the x-axis). Below we dissect the strengths and weaknesses of every method. We start from the baseline, then we jump to "`LinnOS+HL`" (the best outcome), followed by the others.

Baseline. The `Base` lines in Figure 9 confirm unpredictability of flash storage with latencies that spike almost exponentially in between p95 to p99.99, increasing the average latencies to 1.3–6.5 times compared to our best cases

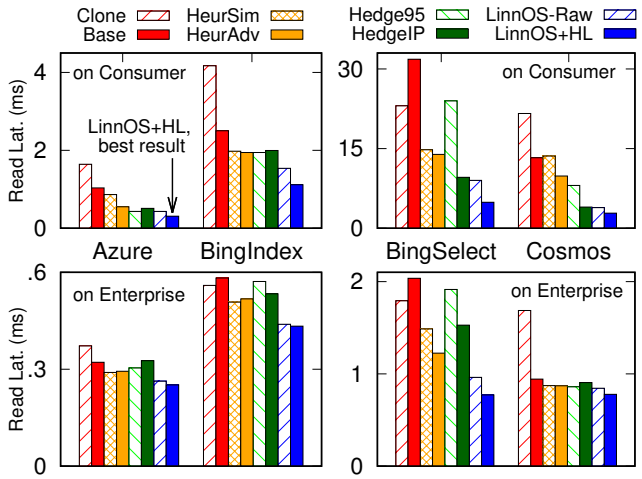


Figure 8: **Average latencies.** *The figures show that LinnOS consistently outperforms all other methods, as explained in Section 5.3. The top and bottom graphs represent experiments on the consumer and enterprise arrays, respectively.*

(Figure 8). Clearly, flash arrays with data redundancy should adopt tail-cutting methods to achieve higher predictability.

LinnOS+HL. This label represents the LinnOS method combined with high-percentile hedging for masking the small inaccuracy that is intrinsically hard to eliminate in a neural network (Section 4.4). That is, to compensate for the inaccuracies that cause false submits, our application sends a duplicate I/O after pX latency time has elapsed, where X is the smaller of 95 and (1 – false submit rate)×100. We use the false submit rates from the training process (Figure 10 in Section 5.4).

[Key outcome] → The average latencies in Figure 8 show that `LinnOS+HL` *consistently outperforms all other methods across different workloads and platforms.* On average, `LinnOS+HL` reduces latency by 9.6-79.6% compared to p95 hedging (`Hedge95`), 14.2-49.5% to hedging with our IP algorithm (`HedgeIP`), and 10.7-71.2% to an advanced heuristic (`HeurAdv`). These speed-ups are a product of the stable latencies; *in Figure 9,* `LinnOS+HL` *lines exhibit stable latencies even at extremely high percentiles, p99 to 99.99.* These results bring a positive conclusion that the downsides of LinnOS (a 15µs failover overhead including a 6µs per-I/O inference cost and the inaccuracies) are outweighed by its effectiveness in delivering predictable latencies.

LinnOS (Raw). Here we show LinnOS efficiency even without hedging (*i.e.*, revoke+failover without I/O duplication). The `LinnOS-Raw` bars in Figure 8 shows that LinnOS by itself is effective enough, only 1.3-45.7% worse than `LinnOS+HL`, and compared to p95 hedging, `LinnOS-Raw` reduces latency by 0.3-62.3%, and to an advanced heuristic, by 3.0-60.7%. Figure 9 details why adding hedging is useful. At high percentiles, above p99, `LinnOS-Raw` starts exhibiting high latencies (due to false submits). Learning from the "small-tail" behavior of hedging (*e.g.*, the `Hedge95` lines), we combined the best of the two in `LinnOS+HL`.

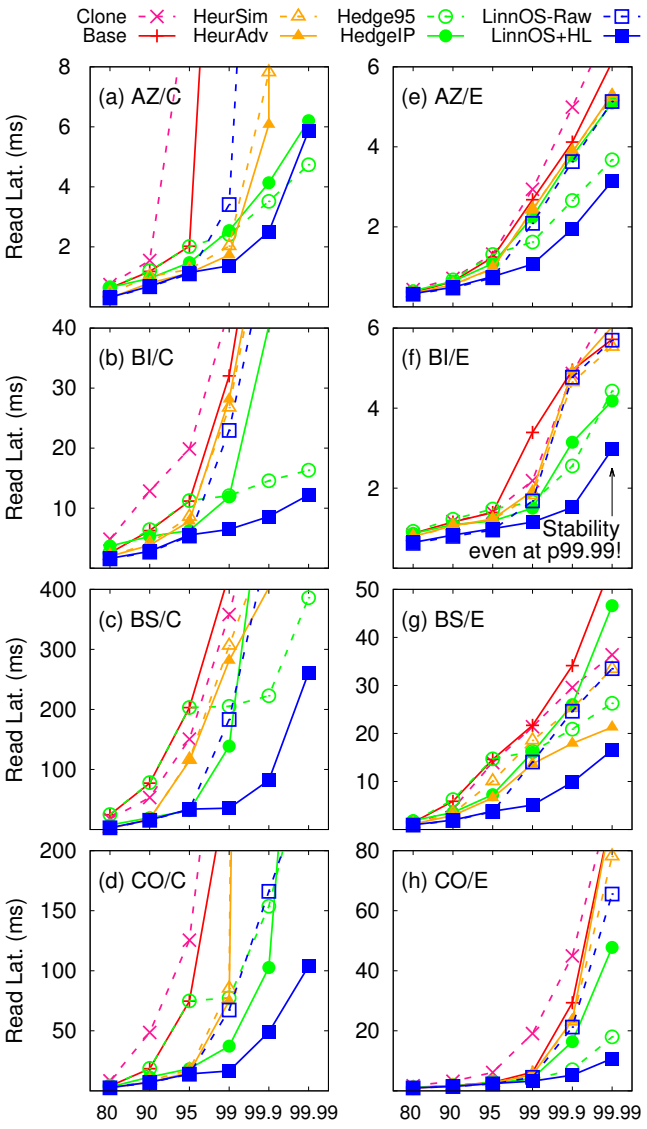


Figure 9: **Percentile latencies.** *Explained in Section 5.3, the figures show that LinnOS +HL delivers the most predictable latencies (y-axis) across all percentiles (x-axis), even at p99.99. In Figure (a), "AZ/C" means Azure running on consumer array.*

Hedging at p95. Sending a duplicate I/O after a p95-latency timeout has elapsed is a popular method used in the field [12, 21]. Figure 9 shows that, in general, this method is effective in cutting latency tail but generally incurs *higher* latencies than `LinnOS+HL`. This is because `Hedge95` needs to *wait* for the timeout to happen before sending the duplicate I/Os, while LinnOS returns a timely revocation that allows the application to failover quickly. As the implication, `Hedge95`, *on average, is slower* than `LinnOS+HL` or even

LinnOS-Raw (Figure 8).

Hedging at IP. Many of the IP values shown in Table 2 are below p95, which raises the question of whether hedging at IP would be better than at p95. The average values in Figure 8 show a mixed result. On the consumer devices, HedgeIP improves upon Hedge95 by 2x for heavy workloads BingS and Cosmos, but loses by up to 15% in light workloads Azure and BingI. Similarly, on enterprise devices, HedgeIP wins in BingS while slightly losing in the others. Upon further investigation, we see that, for example, in consumer devices, Azure and BingI latencies are generally fast (<2 and 10ms respectively, as shown by the y-axis in Figure 9a-b), hence are *sensitive* to the extra load from duplicate I/Os; HedgeIP in our experiments are sending more duplicates than Hedge95. Nevertheless, our experiments show that for most of the workloads, HedgeIP is more effective than Hedge95, hence systems with hedging can adopt our IP algorithm.

Simple heuristic. The first heuristic we wrote, "HeurSim," is based on a popular heuristic for spinning disks: if the device queue length (the number of outstanding I/Os) is larger than a threshold, the incoming I/O should be retried elsewhere [13, 62, 64]. For the threshold, we use a similar method as HedgeIP, but instead of using IP latency value, we use IP queue length. That is, we first profile the queue length distribution during tracing and then select the queue length at the IP percentile as the threshold for revoking. Figure 8 shows that HeurSim only gives a small improvement over the baseline and is far from the best case. In short, it is not smart enough to infer device-internal disruptions.

Advanced heuristic. We extend HeurSim to a more "advanced" heuristic, HeurAdv. For comparison fairness, we reuse the same intuition we had in building LinnOS and apply it to HeurAdv. An additional task that HeurAdv performs is scanning the last N completed I/Os (N=4, same as in LinnOS) and if this history shows a slow I/O ("slow" as defined in §4.2) but with a low queue length (less than the median), it will mark the drive as "internally busy." In this state, incoming I/Os will not be admitted unless the queue length drops to a low value (less than the lower-quartile queue length). The state will not be changed from "busy" to "normal" until it sees recent I/Os become fast ("fast" as defined in §4.2).

[2nd key outcome] → Figure 8 shows that HeurAdv improves upon HeurSim in most cases, but still loses from other methods. We would like to note that we spent several weeks tuning the heuristic to the "best" outcome we can achieve. Continued expansion and tuning of the heuristic is possible. However, the main difficulty that will arise is the *large design space* of parameters (normal/busy states, median and lower-quartile queue lengths, etc.) that must be optimally and *manually configured* for different workloads and devices. This is where we show that machine learning helps. The use of a lightweight neural network allows us to focus on deciding what features matter, but at the same time letting the model learn and reverse-engineer SSD behaviors. In our case, LinnOS neural network *auto-trains all the 8706 weights* for different devices and workloads.

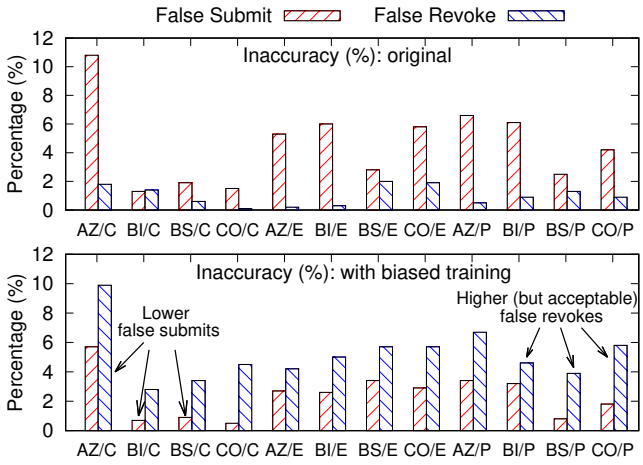


Figure 10: **Low inaccuracy.** *The figure shows the percentage of false submits and false revokes. Note that only false submits really matter (see Section 5.4). Additionally,* **"P"** *represents other device models that we can access from a public cloud. For graph readability, here for "P" we only show the results for one device model, while the observations stand across the rest. In total, the accuracy evaluation covers 10 device models (1C+3E+6P).*

Cloning. This method is essentially p00 hedging, sending a duplicate I/O for every I/O on the outset. Although SSDs are fast and have internal parallelism, Figure 8 shows that Clone is mostly worse than the baseline due to the 2x load.

5.4 (Low) Inaccuracy

We now measure LinnOS inaccuracy by counting the number of false submits and false revokes (Section 4.4). The live experiments can only measure the former but not the latter. This is because revoked I/Os are never submitted to the device, hence we never know whether the revoke is accurate or not. Thus, for this evaluation, we measure inaccuracy in an offline way using TensorFlow, just like the training phase. However, note that both the training and test data were collected from running the workloads on real flash arrays (*i.e.*, not simulated data). Just like before, we use 1-hour data sets for training and then pick three different 1-hour data sets for testing accuracy, and measure the average inaccuracy.

Figure 10 shows the inaccuracies before and after we use biased training. To recap Section 4.4, false submits are more dangerous than false revokes. Without bias, the top graph shows that the false submit rates (red bars) are high, between 1.3% to 10.8%. With biased training, as shown in the bottom graph, we successfully lower the false submit rates to 0.7-5.7%, by *shifting* the inaccuracies to false revokes, which are more tolerable as explained in Section 4.4. For example, let us assume an inferior scenario of p80 inflection point (*i.e.*,

Model:	A	B	C	D	E
Acc. (%)	−(3-12)	−(1-4)	+(1-2)	+(4-5)	+(8-12)
Perf. (µs)	−4	−1	+40	+94	+1670

Table 3: **Trade-offs balance.** *This table is explained Section 5.5. All the +/− of accuracy and performance values are compared to our final neural network model described in §4.*

20% slowness), which means the probability that all three replicas are slow is 0.008 ($(^{20}/_{100})^3$). Thus, although we have spiked up the false revokes to 2.8-9.7% in Figure 10b, *only 0.008* of these false revokes probabilistically will result in slow I/Os. Finally, as mentioned before, for masking the dangerous low inaccuracy (the 0.7-5.7% false submits), combining LinnOS with high-percentile hedging (`LinnOS+HL`) led to a powerful result.

5.5 Trade-offs

Table 3 shows some possible trade-offs between inference overhead and accuracy (models A-E, with accuracy involving both false submits and false revokes). On one hand, if lower overhead is preferred and some accuracy loss is acceptable, then one option is to trim the input features and the model. For example, in model B with R=3 (*i.e.*, including fewer history I/Os instead of R=4) can reduce the number of input features from 31 to 24 and lower inference overhead, −1µs, but it will bring some accuracy loss, −(1-4%), due to fewer inputs. Or, if even lower overhead is favorable, then in model A we can further cut the input features (R=2, 17 features) and use a slimmer hidden layer (from 256 neurons to 128), resulting in a lower inference time,−4µs, while bringing larger accuracy loss, −(3-12%).

If higher accuracy is needed, then we can bring in more features and heavier models. For example, in model C, by adding one more hidden layer to the model, we can gain +(1-2%) higher accuracy, while the inference overhead rises by +40µs. Taking a step further, we can involve more features (up to R=10 and 73 features) and more hidden layers (three layers with 256-512-256 neurons) to push the accuracy gain by +(4-5%), but an increased overhead, +94µs. The extreme model E includes block offsets in the input features (2048 features in total) and applies a model with five hidden layers (with 512 neurons each). For some traces, this model improves the accuracy by +(8-12)%, but its inference overhead, +1670µs, is extremely high for live inference.

5.6 Other Evaluations

5.6.1 Additional Performance Evaluations

Other possible manually-tuned heuristics. To get a sense of how much performance a heuristic can ultimately reach, we pick several 10-min slices from the traces and manually tweak the adjustable parameters of `HeurSim` and `HeurAdv` with various thresholds until an optimal outcome is achieved. In a nutshell, we start with the generic `HeurSim` and `HeurAdv`, evaluate them with the sliced traces, track the high-latency I/Os that are not revoked, update the thresholds to catch these I/Os without causing too many false revokes (*e.g.*, >15%), re-evaluate and repeat the entire process until an approximate optimum is observed. This approach is indeed capable of granting heuristics a further stretch. For example, we see a few cases where tweaked heuristics can outperform `LinnOS-Raw` by up to 20% at p95. However, this tuning procedure is onerous and impractical in real runs as the repeated manual tweaking is too slow to catch up with the fluctuation of incoming workloads.

LinnOS+H99. We also try `LinnOS+H99`, which employs p99 hedging that only generates 1% extra I/Os. Figure 11a shows one of its comparisons with `LinnOS+HL`. Generally, `LinnOS+H99` encounters a larger tail area due to longer waiting, but responds faster at lower percentiles due to less extra I/Os. With that, sometimes `LinnOS+HL` can show slightly worse average latencies (up to 3%) than `LinnOS+H99` (Figure 11b). However, in a large majority of our benchmarks, `LinnOS+HL` achieves 1.7-39.2% better average latencies than `LinnOS+H99`.

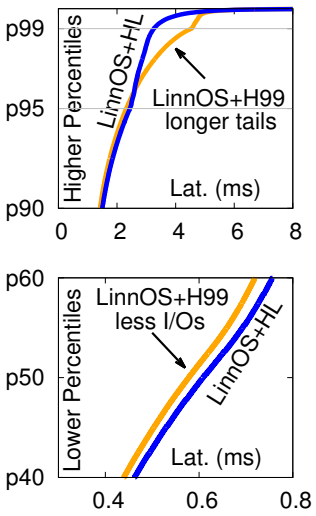


Figure 11: **LinnOS+H99.**

5.6.2 On Public Traces

Beyond our evaluation with Microsoft traces, Figure 12 shows a quick evaluation with the latest SSD traces published on the SNIA website [9] run on our consumer flash array. The result confirms that LinnOS also exhibits low inaccuracy (Figure 12a) and substantial latency improvement (Figure 12b).

5.6.3 MongoDB on Different Filesystems

To see how data applications can benefit from LinnOS, we set up a local MongoDB replica set on top of our three enterprise drives with homogeneous filesystem settings. For each type of filesystem, MongoDB receives 120K random read requests, and all drives run Microsoft traces as background noise when serving MongoDB requests as latency-critical I/Os. Here, we focus on high-percentile latency (*e.g.*, p99 latency) since the average latency is largely impacted by filesystem buffering, while the tail latency reflects the raw performance from the devices.

Figure 13 shows that with LinnOS, MongoDB achieves

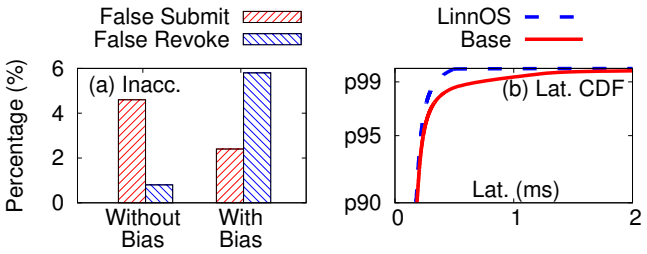


Figure 12: **On public traces.** *As explained in §5.6.2.*

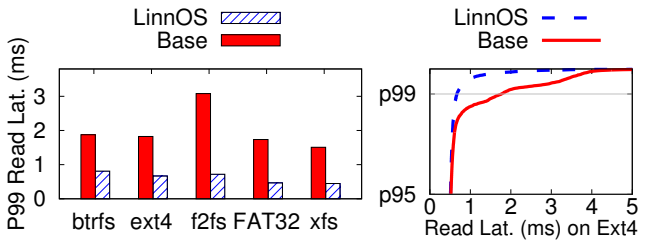


Figure 13: **MongoDB on different filesystems.** *This figure shows that LinnOS can easily help data applications achieve more predictable latency (§5.6.3).*

much more predictable performance. For example, with all underlying devices formatted with f2fs, LinnOS reduces the p99 latency by 76.7%. Moreover, LinnOS only requires minor changes to MongoDB and filesystems: 50 additional LOC. For example, the filesystems should directly return LinnOS's error code to the applications instead of conducting unnecessary self-checking, and MongoDB needs to be slightly modified to reuse its built-in failover logic.

5.6.4 Computation Overhead/Optimization

CPU overhead. A reasonable concern is that if the entire OS has many neural networks, then it will be CPU-intensive. Across all the benchmarks and SSDs, paired with a lightweight neural network, each device only costs 0.3-0.7% of the host CPU resource, making LinnOS practical for large-scale deployments.

Co-processors for acceleration. As mentioned in Section 4.5, additional processors can be utilized to speed up the inference. By utilizing one more CPU core, LinnOS can reduce the inference overhead by 36% (to 4µs), with the maximal CPU usage increased up to 1.4% per device.

6 Conclusion and Discussion

We have presented LinnOS, to the best of our knowledge, the first operating system capable of inferring the speed of every I/O to flash storage. We have shown the feasibility of using a light neural network in the operating system for making frequent, fine-grained, black-box live inferences. LinnOS outperforms many other methods and successfully brings predictability on unpredictable flash storage. We also believe that LinnOS's success leads to exciting discussions and questions that can spur future work:

On performance. Though LinnOS inference overhead (4-6µs) is less noticeable compared with the access latency of current SSDs (*e.g.*, 80µs), it could become problematic as SSDs march to 10µs latency range. Also, the consumption of computation resources can increase substantially as the IOPS grow. How to further lower the inference cost (*e.g.*, to 1µs) to support faster devices and higher throughput? Can advanced accelerators help accelerate OS kernel operations? Can near-storage/data processing help? Can we skip the inference when the outcome is highly assured (*e.g.*, the queue length is very low)? Can we cache the approximation results for popular predictions?

On masking the inaccuracy of machine learning. As machine learning (*e.g.*, LinnOS) can never achieve 100% accuracy, how should "ML-for-system" solutions mask the cases that machine learning fails to catch, while still benefiting from its generality? Is marrying learning and heuristic (*e.g.*, as in `LinnOS+HL`) a powerful option that exploits the advantages of both worlds?

On other integrations and extensions. One interesting question raised by LinnOS is why the latency behavior of SSDs—devices with complex idiosyncrasies—can be learned by the block layer with a few observable features. Understanding this can help other higher layers such as RAID, direct device access (SPDK), user/device-level filesystems, or distributed storage adopt our concept. Likewise, in lower layers, it is also a possibility in the future to have SSDs with latency inference capability built in. Although, arguably, one can say that the device already has full knowledge of its internals and does not need a black-box prediction, an argument can be made that SSD vendors can use the same machine learning method across different internal architectures. Hence, they do not need to re-develop the inference logic every time they modify the internal hardware, logic, and policies. Alternatively, SSD vendors can employ "gray-box" learning that incorporates some of the internal knowledge.

On precision. Can fast/slow inference be converted to a more precise latency inference, such as latency ranges (*e.g.*, 2-4µs, 4-8µs, ...), percentile buckets (*e.g.*, p0-p10, ..., p90-p100), or precise latency with high accuracy? Can model permutation or other machine learning techniques help?

7 Acknowledgments

We thank Tom Anderson, our shepherd, and the anonymous reviewers for their tremendous feedback and helpful comments. We also thank the Azure CSI group for providing the traces. This material was supported by funding from NSF (grant Nos. CCF-2028427, CNS-1405959, CNS-1526304, CNS-1764039, and CNS-1823032), ARO (W911NF1920321), DOE (DESC00141950003) and UChicago CERES Center, as well as generous donations from Dell EMC, Google, and NetApp.

References

[1] Cassandra - Speculative Execution for Reads / Eager Retries. https://issues.apache.org/jira/browse/CASSANDRA-4705.

[2] Chameleon. https://www.chameleoncloud.org.

[3] Ethernet: The High Bandwidth Low-Latency Data Center Switching Fabric. http://www.force10networks.com/whitepapers/pdf/F10_wp_Ethernet.pdf.

[4] GreyBeards on Storage. https://silvertonconsulting.com/gbos2/tag/tail-latency/.

[5] MongoDB - Basic Support for Operation Hedging in NetworkInterfaceTL. https://jira.mongodb.org/browse/SERVER-45432.

[6] New GraphCore IPU BenchMarks. https://www.graphcore.ai/posts/new-graphcore-ipu-benchmarks.

[7] Pareto Distribution. https://en.wikipedia.org/wiki/Pareto_distribution.

[8] Rapid Read Protection in Cassandra 2.0.2. https://www.datastax.com/blog/2013/10/rapid-read-protection-cassandra-202.

[9] SNIA I/O Trace Data Files. http://iotta.snia.org/traces.

[10] The Data Center Flash Storage Market Is Expected to Grow at a CAGR of Nearly About 17% during 2018-2024. https://prn.to/2z58q4L.

[11] The Evolution of Image Classification Explained. https://stanford.edu/~shervine/blog/evolution-image-classification-explained.

[12] Tuning Speculative Retries to Fight Latency. https://www.youtube.com/watch?v=uRJSuQofJWQ, 2016.

[13] Irfan Ahmad. Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2007.

[14] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[15] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the Performance Variation in Modern Storage Stacks. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[16] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[17] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[18] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[19] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[20] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[21] Jeffrey Dean and Luiz Andre Barroso. The Tail at Scale. *Communications of the ACM (CACM)*, 56(2), 2013.

[22] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[23] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[24] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, P. Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[25] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[26] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S.

Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[27] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 2017 EuroSys Conference (EuroSys)*, 2017.

[28] Henry Hoffmann. JouleGuard: energy guarantees for approximate applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[29] Chin-Jung Hsu, Rajesh K Panta, Moo-Ryong Ra, and Vincent W. Freeh. Inside-Out: Reliable Performance Prediction for Distributed Storage Systems in the Cloud. In *The 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016.

[30] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[31] Amber Huffman. Addressing IO Determinism Challenges at Scale with NVM Express Part 2: Renegotiating the Host/Device Contract. In *Proceedings of the 2017 Non-Volatile Memory Workshop (NVMW)*, 2017.

[32] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[33] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA-23)*, 2017.

[34] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Kandemir. Design of a Host Interface Logic for GC-Free SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 8(1), May 2019.

[35] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[36] Bryan S. Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an Autonomic SSD Architecture. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[37] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[38] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *Proceedings of the 17th USENIX Symposium on File and Storage Technologies (FAST)*, 2019.

[39] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. Coordinating Garbage Collection for Arrays of Solid-State Drives. *IEEE Transactions on Computers (TC)*, 63(4), April 2014.

[40] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-state Drives. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[41] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[42] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[43] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[44] Sungjin Lee, Ming Liu, SangWoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed Flash. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[45] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[46] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the 2016 EuroSys Conference (EuroSys)*, 2016.

[47] Chun-Yi Liu, Jagadish B. Kotra, Myoungsoo Jung, Mahmut T. Kandemir, and Chita R. Das. SOML Read: Rethinking the Read Operation Granularity of 3D NAND SSDs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[48] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[49] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[50] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Hiding the Microsecond-Scale Latency of Storage-Class Memories with Duplexity. In *Proceedings of the 2019 Non-Volatile Memory Workshop (NVMW)*, 2019.

[51] Nikita Mishra, Connor Imes, John D. Laferty, and Henry Hoffmann. CALOREE: Learning Control for Predictable Latency and Low Energy. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[52] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. ESP: A Machine Learning Approach to Predicting Application Interference. In *The 14th International Conference on Autonomic Computing (ICAC)*, 2017.

[53] Pulkit A. Misra, Mara F. Borge, Iigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the 2019 EuroSys Conference (EuroSys)*, 2019.

[54] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[55] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[56] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[57] Chris Petersen. Addressing IO Determinism Challenges at Scale with NVM Express. In *Proceedings of the 2017 Non-Volatile Memory Workshop (NVMW)*, 2017.

[58] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[59] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[60] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for OS-level power management. In *Proceedings of the 2009 EuroSys Conference (EuroSys)*, 2009.

[61] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gmez-Luna, and Onur Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[62] Toby J. Teorey and Tad B. Pinkerton. A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM (CACM)*, 15(3), 1972.

[63] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[64] Elizabeth Varki, Arif Merchant, Jianzhang Xu, and Xiaozhou Qiu. Issues and Challenges in the Performance Analysis of Real Disk Arrays. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(6), 2004.

[65] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[66] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage Device Performance Prediction with CART Models. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2004.

[67] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and Inference with Integers in Deep Neural Networks. In *6th International Conference on Learning Representations (ICLR)*, 2018.

[68] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. Overcome the GC-induced Performance Variability in SSD-based RAIDs with Request Redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(5), May 2019.

[69] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. GC-aware Request Steering with Improved Performance and Reliability for SSD-based RAIDs. In *Proceedings of the 32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[70] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Andy Rudoff, and Steven Swanson. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[71] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[72] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet. In *Proceedings of the 2017 ACM on Multimedia Conference (ACMMM)*, 2017.

[73] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[74] Li Yin, Sandeep Uttamchandani, and Randy Katz. An Empirical Exploration of Black-Box Performance Models for Storage Systems. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006.

[75] Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. Guaranteeing Deadlines for Inter-Datacenter Transfers. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[76] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[77] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

A Artifact Appendix

A.1 Abstract

We assemble an executable LinnOS workflow that runs on Chameleon Cloud Research Platform [2]. This self-contained artifact contains the major components and step-by-step instructions.

A.2 Artifact check-list

- **Program:** LinnOS with preprocess scripts [4].
- **Data set:** Example I/O traces.
- **Run-time environment:** Chameleon's shared Jupyter experiment environment.
- **Hardware:** A flash array with at least three SSDs.
- **Output:** Trained models for I/O prediction and latency CDF lines.
- **Experiments:** LinnOS workflow.
- **Expected experiment run time:** Several hours.
- **Public link:** https://www.chameleoncloud.org/experiment/share/15?s=409ab137f20e4cd38ae3dd4e0d4bfa7c

A.3 Description

A.3.1 How to access

Access the public link provided above and click the "Launch on Chameleon" botton (account required to access Chameleon resources), then see Readme.txt for a high-level description and LinnOS.ipynb for step-to-step instructions.

A.3.2 Hardware dependencies

Evaluating LinnOS requires a flash array with at least three SSDs, which are provided by the storage-hierarchy instances from Chameleon Testbed.

A.3.3 Data sets

The artifact contains some example I/O traces, which are used in the workflow for testing purposes.

A.4 Installation

Step-by-step installation instructions are available in the artifact.

A.5 Evaluation and expected result

Upon successful running, the workflow should produce a trained model, the accuracy outcome, and the I/O latency distribution of LinnOS and baseline. Please see readme.txt in the artifact for further details.

[4] Excluding the data collection and analysis code that may reveal sensitive information in Microsoft traces



A large scale analysis of hundreds of in-memory cache clusters at Twitter

Juncheng Yang
Carnegie Mellon University

Yao Yue
Twitter

K. V. Rashmi
Carnegie Mellon University

Abstract

Modern web services use in-memory caching extensively to increase throughput and reduce latency. There have been several workload analyses of production systems that have fueled research in improving the effectiveness of in-memory caching systems. However, the coverage is still sparse considering the wide spectrum of industrial cache use cases. In this work, we significantly further the understanding of real-world cache workloads by collecting production traces from 153 in-memory cache clusters at Twitter, sifting through over 80 TB of data, and sometimes interpreting the workloads in the context of the business logic behind them. We perform a comprehensive analysis to characterize cache workloads based on traffic pattern, time-to-live (TTL), popularity distribution, and size distribution. A fine-grained view of different workloads uncover the diversity of use cases: many are far more write-heavy or more skewed than previously shown and some display unique temporal patterns. We also observe that TTL is an important and sometimes defining parameter of cache working sets. Our simulations show that ideal replacement strategy in production caches can be surprising, for example, FIFO works the best for a large number of workloads.

1 Introduction

In-memory caching systems such as Memcached [14] and Redis [16] are heavily used by modern web applications to reduce accesses to storage and avoid repeated computations. Their popularity has sparked a lot of research, such as reducing miss ratio [26, 28, 36, 37], or increasing throughput and reducing latency [43, 52, 53, 56]. On the other hand, the effectiveness and performance of in-memory caching can be workload dependent. And several important workload analyses against production systems [24, 59] have guided the explorations of performance improvements with the right context and tradeoffs in the past decade [43, 56].

Nonetheless, there remains a significant gap in the understanding of current in-memory caching workloads. Firstly, there has been a lack of comprehensive studies covering the wide range of use cases in today's production systems. Secondly, there have been new trends in in-memory caching usage since the publication of previous work [24]. Thirdly, some aspects of in-memory caching received little attention in the existing studies, but are known as critical to practitioners. For example, TTL is an important aspect of configuring in-memory caching, but it has largely been overlooked in research. Last but not least, unlike other areas where open-source traces [62, 63, 68, 70] or benchmarks [38] are available, there has been a lack of open-source in-memory caching traces. Researchers have to rely on storage caching traces [26], key-value database benchmarks [43, 56] or synthetic workloads [39, 57] to evaluate in-memory caching systems. Such sources either have different characteristics or do not capture all the characteristics of production in-memory caching workloads. For example, key-value database benchmarks and synthetic workloads don't consider how object size distribution changes over time, which impacts both miss ratio and throughput of in-memory caching systems.

In this work, we bridge this gap by collecting and analyzing workload traces from 153 Twemcache [6] clusters at Twitter, one of the most influential social media companies known for its real-time content. Our analysis sheds light on several vital aspects of in-memory caching overlooked in existing studies and identifies areas that need further innovations. The traces used in this paper are made available to the research community [1]. To the best of our knowledge, this is the first work that studied over 100 different cache workloads covering a wide range of use cases. We believe these workloads are representative of cache usage at social media companies and beyond, and hopefully provide a foundation for future caching system designs. Here's a summary of our discoveries:

1. In-memory caching does not always serve read-heavy workloads, write-heavy (defined as write ratio > 30%) workloads are very common, occurring in more than 35% of the 153 cache clusters we studied.
2. TTL must be considered in in-memory caching because it limits the effective (unexpired) working set size. Efficiently removing expired objects from cache needs to be prioritized over cache eviction.
3. In-memory caching workloads follow approximate Zipfian popularity distribution, sometimes with very high skew. The workloads that show the most deviations tend to be write-heavy workloads.
4. The object size distribution is not static over time. Some workloads show both diurnal patterns and experience sudden, short-lived changes, which pose challenges for slab-based caching systems such as Memcached.

5. Under reasonable cache sizes, FIFO often shows similar performance as LRU, and LRU often exhibits advantages only when the cache size is severely limited.

These findings provide a detailed new look into production in-memory caching systems, while unearthing some surprising aspects not conforming to the folklore and to the commonly used assumptions.

2 In-memory Caching at Twitter

2.1 Service Architecture and Caching

Twitter started its migration to a service-oriented architecture, also known as microservices, in 2011 [8]. Around the same time, Twitter started developing its container solution [2, 3] to support the impending wave of services. Fast forward to 2020, the real-time serving stack is mostly service-oriented, with hundreds of services running inside containers in production. As a core component of Twitter's infrastructure, in-memory caching has grown alongside this transition. Petabytes of DRAM and hundreds of thousands of cores are provisioned for caching clusters, which are containerized.

At Twitter, in-memory caching is a managed service, and new clusters are provisioned semi-automatically to be used as look-aside cache [59] upon request. There are two in-memory caching solutions deployed in production, Twemcache, a fork of Memcached [14], is a key-value cache providing high throughput and low latency. The other solution, named Nighthawk, is Redis-based and supports rich data structures and replication for data availability. In this work, we focus on Twemcache because it serves the majority of cache traffic.

Cache clusters at Twitter are considered *single-tenant*[1] based on the service team requesting them. This setup is very beneficial to workload analysis, because it allows us to tag use cases, collect traces, and study the properties of workloads individually. A multi-tenant setup will make similar study extremely difficult, as researchers have to tease out individual workloads from the mixture, and somehow connect them to their use cases. In addition, smaller but distinct workloads can easily be overlooked or mis-characterized due to low traffic.

Unlike other cache cluster deployments, such as social graph caching [19, 30] or CDN caching [47, 69], Twemcache is mostly deployed as a single-layer cache, which allows us to analyze the requests directly from clients without being filtered by other caches. Previous work [47] has shown that layering has an impact on properties of caching workloads, such as popularity distribution. This single-tenant, single-layer design provides us the perfect opportunity to study the properties of the workloads.

2.2 Twemcache Provisioning

There are close to 200 Twemcache clusters in each data center as of writing. Twemcache containers are highly homogeneous and typically small, and a single host can run many of them. The number of instances provisioned for each cache cluster is computed from user inputs including throughput, estimated dataset sizes, and fault tolerance. The number of instances of each cluster is automatically calculated first by identifying the correct bottleneck and then applying other constraints, such as number of connections to support. Size of production cache clusters ranges from 20 to thousands of instances.

[1] Although each cluster is single-tenant, each tenant might cache multiple types of objects of different characteristics.

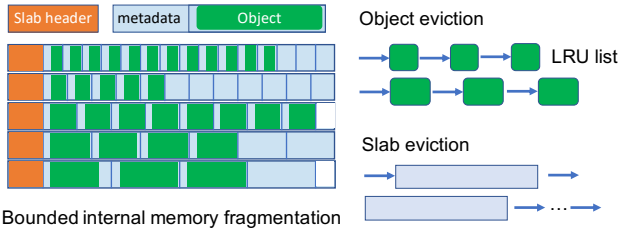


Figure 1: Slab-based memory management for bounded memory fragmentation. While Memcached uses object eviction, Twemcache uses slab eviction, which evicts all objects in one slab and returns the slab to global pool.

2.3 Overview of Twemcache

Twemcache forked an earlier version of Memcached with some customized features. In this section, we briefly describe some of the key aspects of its designs.

Slab-based memory management Twemcache often stores small and variable-sized objects in the range of a few bytes to 10s of KB. On-demand heap memory allocators such as ptmalloc [45], jemalloc [13] can cause large and unbounded external memory fragmentation in such a scenario, which is highly undesirable in production environment, especially when using smaller containers. To avoid this, Twemcache inherits the slab-based memory management from Memcached (Figure 1). Memory is allocated as fixed size chunks called *slabs*, which default to 1 MB. Each slab is then evenly divided into smaller chunks called *items*. The *class* of each slab decides the size of its items. By default, Twemcache grows item size from a configurable minimum (default to 88 bytes) to just under a whole slab. The growth is typically exponential, controlled by a floating point number called growth factor (default to 1.25), though Twemcache also allows precise configuration of specific item sizes. Higher slab classes correspond to larger items. An object is mapped to the slab class that best fits it, including metadata. In Twemcache, this per-object metadata is 49 bytes. By default, a slab of class 12 has 891 items of 1176 bytes each, and each item stores up to 1127 bytes of key plus value. Slab-based allocator eliminates external memory fragmentation at the cost of bounded internal memory fragmentation.

Eviction in slab-based cache To store a new object, Twemcache first computes the slab class by object size. If there is a slab with at least one free item in this slab class, Twemcache uses the free item. Otherwise, Twemcache tries to allocate a new slab into this class. When memory is full,

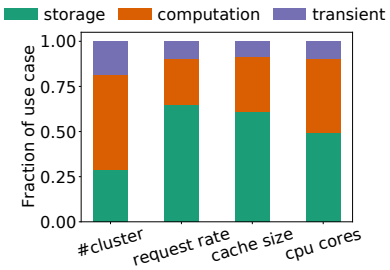


Figure 2: Resources consumed for the three cache use cases.

slab eviction is needed for allocation.

Some caching systems such as Memcached primarily performs item-level eviction, which happens in the same slab class as the new object. Memcached uses an approximate LRU queue per slab class to track and evict the least recently used item. This works well as long as object size distribution remains static. However, this is often not true in reality. For example, if all keys start with small values that grow over time, new writes will eventually require objects to be stored in a higher slab class. However, if all memory has been allocated when this happens, there will be effectively no memory to give out. This problem is called *slab calcification* and is further explored in Section 4.6.2. Memcached developed a series of heuristics to move memory between slab classes, and yet they have been shown as non-optimal [10, 11, 17, 46] and error prone [9].

To avoid slab calcification, Twemcache uses slab eviction only (Figure 1). This allows the evicted slab to transition into any other slab class. There are three approaches to choose the slab to evict: choosing a slab randomly (random slab), choosing the least recently used slab (slabLRU), and choosing the least recently created slab (slabLRC). In addition to avoiding slab calcification, slab-only eviction removes two pointers from object metadata compared to Memcached. We further compare object eviction and slab eviction in Section 6.

2.4 Cache Use Cases

At Twitter, it is generally recognized that there are three main use cases of Twemcache: caching for storage, caching for computation, and caching for transient data. We remark that there is no strict boundary between the three categories, and production clusters are not explicitly labeled. Thus the percentages given below are rough estimates based on our understanding of each cache cluster and their corresponding application.

2.4.1 Caching for Storage

Using cache to facilitate reading from storage is the most common use case. Backend storage such as databases usually has a longer latency and a lower bandwidth than in-memory cache. Therefore, caching these objects reduce access latency, increases throughput, and shelters the backend from excessive read traffic. This use case has received the most attention in research. Several efforts have been devoted to reducing miss ratio [26–28, 36, 37, 41, 47, 72] , redesigning for a denser storage device to fit larger working sets [19, 42, 65], improving load balancing [33, 34, 39] and increasing throughput [43, 56].

As shown in Figure 2, although only 30% of the clusters fall into this category, they account for 65% of the requests served by Twemcache, 60% of the total DRAM used, and 50% of all CPU cores provisioned.

2.4.2 Caching for Computation

Caching for computation is not new — using DRAM to cache query results has been studied and used since more than two decades ago [20, 58]. As real-time stream processing and machine learning (ML) become increasingly popular, an increasing number of cache clusters are devoted to caching computation related data, such as features, intermediate and final results of ML prediction, and so-called object hydration, — populating objects with additional data, which often combines storage access and computation.

Overall, caching for computation accounts for 50% of all Twemcache clusters in cluster count, 26%, 31% and 40% of request rate, cache sizes and CPU cores.

2.4.3 Transient data with no backing store

The third typical cache usage evolves around objects that only live in cache, often for short periods of time. It is not caching in the strict sense, and therefore has received little attention. Nonetheless, in-memory caching is often the only production solution that meets both the performance and scalability requirements of such use cases. While data loss is still undesirable, these use cases really prize speed, and tolerate occasional data loss well enough to work without a fallback.

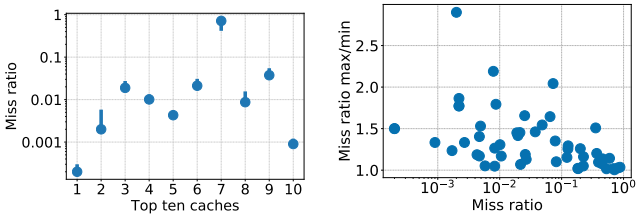
Some notable examples are rate limiters, deduplication caches, and negative result caches. Rate limiters are counters associated with user activities. They track and cap user requests in a given time window and prevent denial-of-service attacks. Deduplication caches are a special case of rate limiters, where the cap is 1. Negative result caches store keys from a larger database that are known to be misses against a smaller, sparsely populated database. These caches short-circuit most queries with negative results, and drastically reduce the traffic targeting the smaller database.

In our measurements, 20% of Twemcache clusters are under this category. Their request rates and cache sizes account for 9% and 8% of all Twemcache request rates and cache sizes, meanwhile, they account for 10% of all CPU cores of Twemcache clusters.

3 Methodology

3.1 Log Collection

Twemcache has a built-in non-blocking request logging utility called `klog` that can keep up with designed throughput in production. While it logs one out of every 100 requests by default, we dynamically changed the sampling ratio to 100% and collected week-long *unsampled* traces from two instances of each Twemcache cluster. Collecting unsampled



(a) Production miss ratio **(b)** Miss ratio variation

Figure 3: a) Production miss ratio of the top ten Twemcache clusters ranked by request rates, the bar shows the max and min miss ratio across one week. Note that the Y-axis is in log scale. b) The ratio between max and min miss ratio is small for most caches.

traces allows us to avoid drawing potentially biased conclusions caused by sampling. Moreover, we chose to collect traces from two instances instead of one to prevent possible cache failure during log collection and to compare results between instances for higher fidelity. Barring cache failures, the two instances have no overlapping keys.

3.2 Log Overview

We collected around 700 billion requests (80 TB in raw file size) from 306 instances of 153 Twemcache clusters, which include all clusters with per-instance request rate more than 1000 queries-per-sec (QPS) at the time of collection. To simplify our analysis and presentation, we focused on the 54 largest caches, which account for 90% of aggregated QPS and 76% of allocated memory. In the following sections, we use Twemcache workloads to refer to the workloads from these 54 Twemcache clusters. Although we only present the results of these 54 caches, we did perform the same analysis on the smaller caches, and they don't change our conclusions.

4 Production Stats and Workload Analysis

In this section, we start by describing some common production metrics to provide a foundation for our discussion, and then move on to workload analyses that can only be performed with detailed traces.

4.1 Miss Ratio

Miss ratio is one of the key metrics that indicate the effectiveness of a cache. Production in-memory caches usually operate at a low miss ratio with small miss ratio variation.

We present the miss ratios of the top ten Twemcache clusters ranked by request rates in Figure 3a where the dot shows the mean miss ratio over a week, and the error bars show the minimum and maximum miss ratio. Eight out of the ten Twemcache clusters have a miss ratio lower than 5%, and six of them have a miss ratio close to or lower than 1%. The only exception is a write-heavy cache cluster, which has a miss ratio of around 70% (see Section 4.3.2 for details about write-heavy workloads). Compared to CDN caching [47], in-memory caching usually has a lower miss ratio.

Besides a low miss ratio, miss ratio stability is also very important. In production, it is the highest miss ratio (and

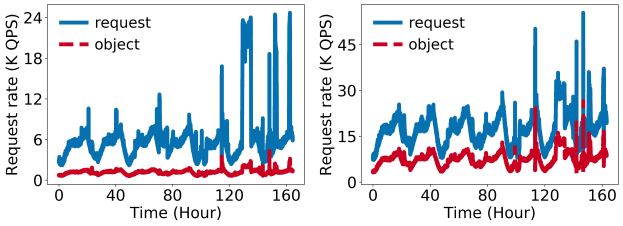


Figure 4: The number of requests and objects being accessed every second for two cache nodes.

request rate) that decides the QPS requirement of the backend. Therefore, a cache with a low miss ratio most of the time, but sometimes a high miss ratio is less useful than a cache with a slightly higher but stable miss ratio. Figure 3b shows the ratios of $\frac{mr_{max}}{mr_{min}}$ over the course of a week for different caches, where mr stands for miss ratio. We observe that most caches have this ratio lower than 1.5. In addition, the caches that have larger ratios usually have a very low miss ratio.

Low miss ratios and high stability in general illustrate the effectiveness of production caches. However, extremely low miss ratios tend to be less robust, which means the corresponding backends have to be provisioned with more margins. Moreover, cache maintenance and failures become a major source of disruption for caches with extremely low miss ratios. The combination of these factors indicate there's typically a limit to how much cache can reduce read traffic or how little traffic backends need to provision for.

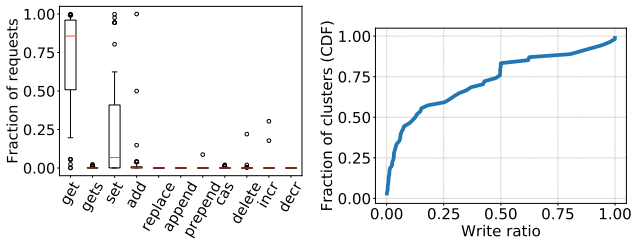
4.2 Request Rate and Hot Keys

Similar to previously observed [24], request rates show diurnal patterns (Figure 4). Besides, spikes in request rate are also very common because cache is the first responder to any change from the frontend services and end users.

When a request rate spike happens, a common belief is that hot keys cause the spikes [33, 48]. Indeed, load spikes often are the results of hot keys. However, we notice it is not always true. As shown in Figure 4, at times, when the request rate (top blue curve) spikes, the number of objects accessed in the same time interval (bottom red curve) also has a spike, indicating that the spikes are triggered by factors other than hot keys. Such factors include client retry requests, external traffic surges, scan-like accesses, and periodic tasks.

In addition to request rate spikes, caches often show other irregularities. For example, in Section 4.6.2, we show that it is common to see sudden changes in object size distribution. These irregularities can happen for various reasons. For instance, users change their behavior due to a social event, the frontend service adds a new feature (or bug), or an internal load test is started.

As a critical component in the infrastructure, caches stop most of the requests from hitting the backend, and they should be designed to tolerate these workload changes to absorb the impact.



(a) Relative use of each operation **(b)** Write ratio

Figure 5: a) Ratio of operation in each Twemcache cluster, box shows the 25^{th} and 75^{th} percentile, red bar inside the box shows the mean ratio, and whiskers are 10^{th} and 90^{th} percentile. b) write ratio distribution CDF across Twemcache clusters.

4.3 Types of Operations

Twemcache supports eleven different operations, of which `get` and `set` are the most heavily used by far. In addition, write-heavy cache workloads are very common at Twitter.

4.3.1 Relative usage comparison

We begin from the operations used by Twemcache workloads. Twemcache supports eleven operations `get`, `gets`, `set`, `add`, `cas` (check-and-set), `replace`, `append`, `prepend`, `delete`, `incr` and `decr`[2]. As shown in Figure 5a, `get` and `set` are the two most common operations, and average `get` ratio is close to 90% indicating most of the caches are serving read-heavy workloads. Apart from `get` and `set`, operations `gets`, `add`, `cas`, `delete`, `incr` are also frequently used in Twemcache clusters. However, compared to `get` and `set`, these operations usually account for a smaller percentage of all requests. Nonetheless, these operations serve important roles in in-memory caching. Therefore, as suggested by the author of Memcached, they should not be ignored [15].

4.3.2 Write ratio

Although most caches are read dominant, Figure 5a shows that both `get` and `set` ratios have a large range across caches. We define a workload as write-heavy if the percentage sum of `set`, `add`, `cas`, `replace`, `append`, `prepend`, `incr` and `decr` operations exceeds 30%. Figure 5b shows the distribution of write ratio across caches. More than 35% of all Twemcache clusters are write-heavy, and more than 20% have a write ratio higher than 50%. In other words, in addition to the well-known use case of serving read-heavy workloads, a substantial number of Twemcache clusters are used to serve write-heavy workloads. We identify the main use cases of write-heavy caches below.

Frequently updated data Caches under this category mostly belong to cache for computation or transient data (Section 2.4.2 & 2.4.3). Updates are accumulated in cache before they get persisted, or the keys eventually expire.

[2]See https://github.com/memcached/memcached/wiki/Commands for details about each command.

Opportunistic pre-computation Some services continuously generate data for potential consumption by itself or other services. One example is the caches storing recent user activities, and the cached data are read when a query asks for recent events from a particular user. Many services choose not to fetch relevant data on demand, but instead opportunistically pre-compute them for a much larger set of users. This is feasible because pre-computation often has a bounded cost, and in exchange read queries can be quickly fulfilled by pre-computed results partially or completely. Since this is a trade-off mainly for user experience, the caches under this category see objects with fewer reuse. Therefore, the write ratio is often higher (>80%), and object access (read+write) frequency is often lower. In one case, we saw one cluster with a mean object frequency close to 1.

4.4 TTL

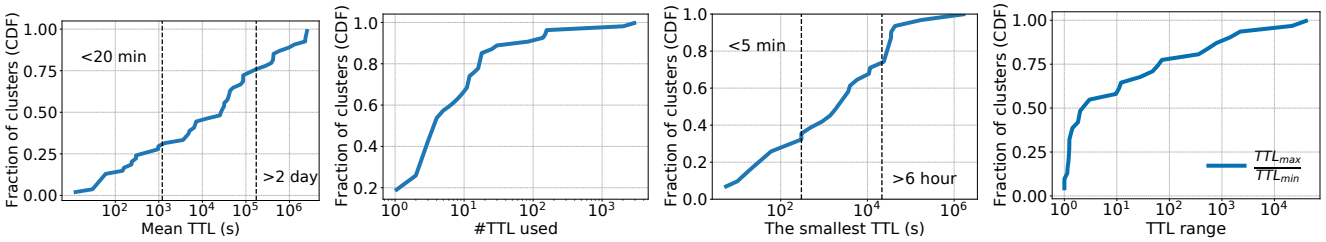
Two important features that distinguish in-memory caching from a persistent key-value store are TTL and cache eviction. While evictions have been widely studied [26, 28], TTL is often overlooked. Nonetheless, TTL has been routinely used in production. Moreover, as a response to GDPR [5], the usage of caching TTL has become mandatory at Twitter to enforce data retention policies. TTL is set when an object is first created in Twemcache, and decides its expiration time. Request attempts to access an expired object will be treated as misses, so keeping expired objects in the cache is not useful.

We observe that in-memory caching workloads often use short TTLs. This usage comes from the dynamic nature of cached objects and the usage for implicit deletion. Under this condition, effectively and efficiently removing expired objects from the cache becomes necessary and important, which provides an alternative to eviction in achieving low miss ratios.

4.4.1 TTL Usages

We measure the mean TTLs used in each Twemcache cluster and show the TTL distribution in Figure 6a. The figure shows that TTL ranges from minutes to days. More than 25% of the workloads use a mean TTL shorter than twenty minutes, and less than 25% of the workloads have a mean TTL longer than two days. Such a TTL range is longer than DNS caching (minutes) [51], but shorter than common CDN object caching (days to weeks). If we divide caches into *short-TTL caches* (TTL $\leq$ 12 hours) and *long-TTL caches* (TTL > 12 hours). Figure 6a shows 66% of all Twemcache clusters have a short mean TTL.

In addition to mean TTL distribution, we have also measured the number of TTL used in each cache. Figure 6b shows that only 20% of the Twemcache workloads use a single TTL, while the rest majority use more than one TTL. In addition, we observe that over 30% of the workloads use more than ten TTLs and there are a few workloads using more than 1000 TTLs. In the last case, some clients intentionally scatter TTLs over a pre-defined time range to avoid objects expiring at the



(a) Mean TTL distribution **(b)** Number of TTL in each cache **(c)** The smallest TTL distribution **(d)** TTL range distribution

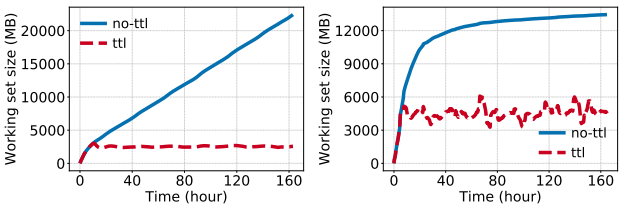
Figure 6: a) More than half of caches have mean TTL shorter than one day. b) Only 20% of caches use single TTL. c) The smallest TTL in each cache can be very long. d) TTLs ranges in workloads are often large.

same time. This technique is called *TTL jitter*. In another case, the clients seek the opposite effect — computing TTLs so that a group of objects will expire at the same, predetermined time.

Besides the number of TTLs used, the smallest TTL and the TTL range, defined as the ratio between TTL_{max} and TTL_{min}, are also important for designing algorithms that remove expired objects (see Section 7). Figure 6c shows that the smallest TTL in each cache varies from 10s of seconds to more than half day. In detail, around 30 to 35% of the caches have their smallest TTL shorter than 300 seconds, and over 25% of caches have the smallest TTL longer than 6 hours. Figure 6d shows the CDF of each workload's TTL range. We observe that fewer than 40% of the workloads have a relatively small TTL range ($< 2\times$ difference), while almost 25% of the caches have $\frac{TTL_{max}}{TTL_{min}}$ over 100.

Below we present the three main purposes of TTL to better explain how TTL settings relate to the usages of the caches.

Bounding inconsistency Objects stored in Twemcache can be highly dynamic. Because cache updates are best-effort, and failed cache writes are not always retried, it is possible that objects stored in in-memory cache are stale. Therefore, applications often use TTL to bound inconsistency, which is also suggested in the AWS Redis documentation [7]. TTLs for this purpose usually have relative large values, in the range of days. Some Twitter services further developed *soft TTL* to achieve a better tradeoff between data consistency and avaiilability. The main idea of soft TTL is to store an additional, often shorter TTL as part of the object value. When application decodes the value of a cached object and notices that the soft TTL has expired, it will refresh the cached value from its corresponding source of truth in the background. Meanwhile, the application continues to use the older value to fulfill current requests without waiting. Soft TTL is typically designed to increase with each background refresh, based on the assumption that newly created objects are more likely to see high volume of updates and therefore inconsistency.



(a) **(b)**

Figure 7: The working set size grows over time when TTL is not considered. However, when TTL is considered, the working set size is capped.

Implicit deletion In some caches, TTL reflects the intrinsic life span of stored objects. One example is the counters used for API rate limiting, which are declared as maximum number of requests allowed in a time window. These counters are typically stored in cache only, and their TTLs match the time windows declared in the API specification. In addition to rate limiters, GDPR required TTL would also fall into this category, so no data would live in cache beyond the duration permitted under the law.

Periodic refresh TTL is also used to promote data freshness. For example, a service that calculates how much a user's interest matches a cluster/community using ML models can make "who-to-follow" type of recommendations with the results. The results are cached for a while because user characteristics tend to be stable in the very short term, and the calculation is relatively expensive. Nonetheless, as users engage with the site, their portraits can change over time. Therefore such a service tends to recompute the results for each user periodically, using or adding the latest data since last update. In this case, TTL is used to pace a relatively expensive operation that should only be performed infrequently. The exact value of the TTL is the result of a balance between computational resources and data freshness, and can often be dynamically updated based on circumstances.

4.4.2 Working Set Size and TTL

Having the majority of caches use short TTLs indicate that the *effective working set size* (WSS_E) — the size of all unexpired objects should be loosely bounded. In contrast, the *total working set size* (WSS_T), the size of all active objects regardless of TTL, can be unbounded.

In our measurements, we identify two types of workloads

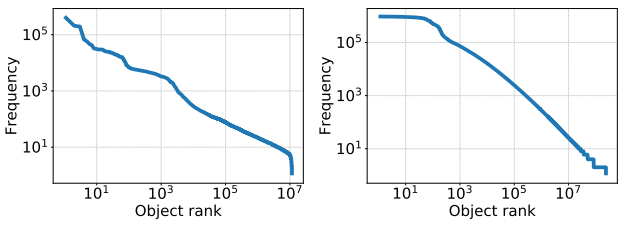
shown in Figure 7. The first type (Figure 7a) has a continuously growing WSS_T, and it is usually related to user-generated content. With new content being generated every second, the total working set size keeps growing. The second type of workload has a large growth rate in WSS_T at first, and then the growth rate decreases after this initial fast-growing period, as shown in Figure 7b. This type of workloads can be users related, the first quick increase corresponds to the most active users, the slow down corresponds to less active users. Although the two workloads show different growth patterns in total working set size, the effective working set size of both arrive at a plateau after reaching its TTL. Although the WSS_E may fluctuate and grow in the long term, the growth rate is much slower compared to WSS_T.

Bounded WSS_E means that, for many caches, there exists a cache size that the cache can achieve compulsory miss ratio, if an in-memory caching system can remove expired objects in time. This suggest the importance of quickly removing expired object from cache, especially for workloads using short TTLs. Unfortunately, while eviction has been widely studied [26, 28, 54], expiration has received little attention. And we will show in Section 7.2, existing solutions fall short on expiration.

4.5 Popularity Distribution

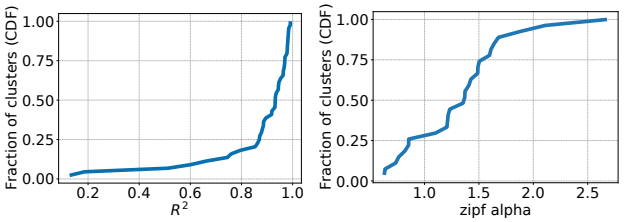
Object popularity is another important characteristic of a caching workload. Popularity distribution is often used to describe the cachebility of a workload. A popular assumption is that cache workloads follow Zipfian distribution [29], and the frequency-rank curve plotted in log-log scale is linear. A large body of work optimizes system performance under this assumption [33, 39, 44, 50, 57, 61]. However, a recent work from Facebook [19] suggested that in-memory caching workloads may not follow Zipfian distribution. Here we present the popularity of the caching workloads at Twitter.

Measuring all Twemcache workloads, we observe majority of the cache workloads still follow Zipfian distribution. However, some workloads show deviations in two ways. First, unpopular objects appear significantly less than expected (Figure 8a) or the most popular objects are less popular than expected (Figure 8b). The first deviation happens when objects



(a) **(b)**

Figure 8: Some workloads showing small deviations from Zipfian popularity. a) The least popular objects are less popular than expected. b) The most popular objects are less popular than expected.



(a) R^2 in Zipfian fitting **(b)** Zipfian parameter α

Figure 9: a) Most of workloads follow Zipfian popularity distribution with large confidence R^2. b) The parameter α in Zipfian distribution is large, and the popularity of most workloads are highly skewed ($\alpha > 1$).

are always accessed multiple times so that there are few objects with frequency smaller than some value. The second deviation happens when the client has an aggressive client-side caching strategy so that the most popular objects are often cached at client. In this case, the cache is no longer single-layer.

Although these deviations happen, they are rare, and we believe it is still reasonable to assume in-memory caching workloads follow Zipfian distribution. Since most part of the frequency-rank curves are linear in the log-log scale, we use linear fitting[3] confidence R^2 [12] as the metric for measuring the goodness of fit. Figure 9a shows the results of fitting. 80% of all workloads have R^2 larger than 0.8, and more than 50% of workloads have R^2 larger than 0.9. These results indicate that the popularity of most in-memory caching workloads at Twitter follows Zipfian distribution. We further measure the parameter α of the Zipfian distribution shown in Figure 9b. The figure shows that most of the α values are in the range from 1 to 2.5, indicating the workloads are highly skewed.

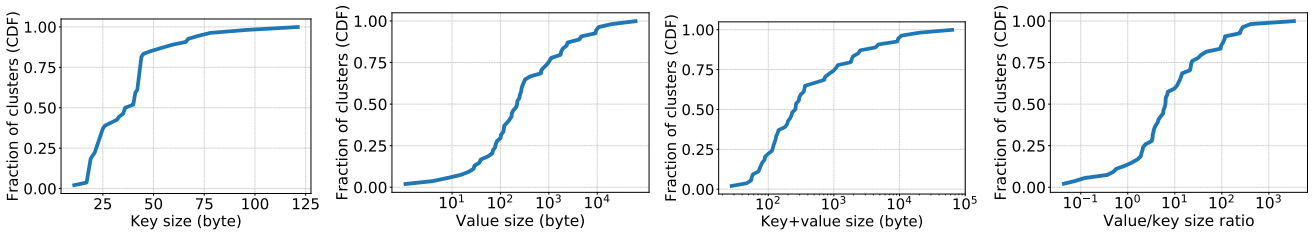
4.6 Object Size

One feature that distinguishes in-memory caching from other types of caching is the object size distribution. We observe that similar to previous observations [24], the majority of objects stored in Twemcache are small. In addition, size distribution is not static over time, and both periodic distribution shifts and sudden changes are observed in multiple workloads.

4.6.1 Size Distribution

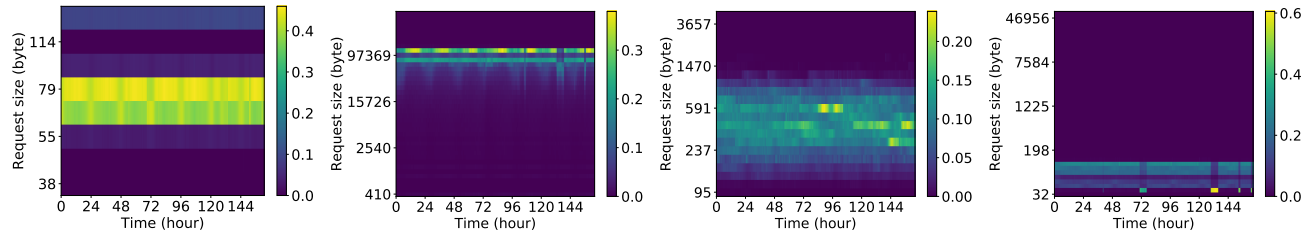
We measure the mean key size and value size in each Twemcache cluster, and present the CDF of the distributions in Figure 10. Figure 10a shows that around 85% of Twemcache clusters have a mean key size smaller than 50 bytes, with a median smaller than 38 bytes. Figure 10b shows that the mean value size falls in the range from 10 bytes to 10 KB, and 25% of workloads show value size smaller than 100 bytes, and median is around 230 bytes. Figure 10c shows that CDF

[3] We remark that linear regression is not the correct way to modelling Zipf distribution from the view of statistics, we perform this to align with existing works [29].



(a) Key size **(b)** Value size **(c)** Object size **(d)** Value/key size ratio

Figure 10: Mean key, value, object size distribution and mean $\frac{value}{key}$ size ratio across all caches.



(a) **(b)** **(c)** **(d)**

Figure 11: Heatmap showing request size distribution over time for four typical caches. X-axis is time, Y-axis is the object size using slab class size as bins, and the color shows the fraction of requests that fall into a slab class in that time window.

distribution of the mean object size (key+value), which is very close to the value size distribution except at small sizes. Value size distribution starts at size 1, while object size distribution starts from size 16. This indicates that for some of the caches, value size is dramatically smaller than the key size. Figure 10d shows the ratio of mean value and key sizes. We observe that 15% of workloads have the mean value size smaller than or equal to the mean key size, and 50% of workloads have value size smaller than 5× key size.

4.6.2 Size Distribution Over Time

In the previous section, we investigated the static size distribution of all objects accessed in the one week's time of each Twemcache cluster. However, the object size distribution of workloads are usually not static over time. In Figure 11, we show how the size distribution changes over time. The X-axis shows the time, and the Y-axis shows the size of objects (using slab class size as bins), the color shows how much of the objects in one time window fall into each slab class. We observe that some of the workloads show diurnal patterns (Figure 11a, 11b), while others show changes without strict patterns.

Periodic/diurnal object size shifts can come from the following sources, a) value for the same key grows over time. and b) size distribution correlates with temporal aspects of key access. For example, text content generated by users in Japan are shorter/smaller than those by users in Germany. In this case, it is the geographical locality that drives the temporal pattern. On the other hand, we do not yet have a good understanding of how most sudden, non-recurring changes happen. Current guesses include user behavior changes during events, and a temporary change in production settings.

Both short-term and long-term size distribution shifts pose additional challenges to memory management in caching systems. They make it hard to control or predict external fragmentation in caches that use heap memory allocators directly, such as Redis. For slab-based caching systems, they can cause slab calcification. In Section 7.5, we discuss why existing techniques do not completely address the problem.

Table 1: Correlation between write ratio and other properties

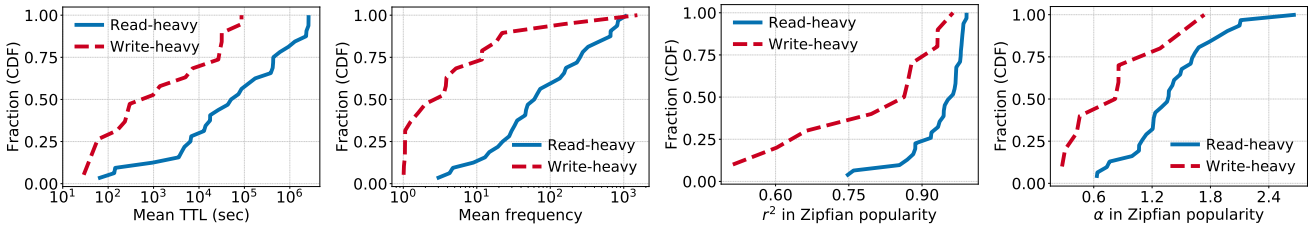
Property	Pearson coefficient with write ratio
log(TTL)	-0.6336
log(Frequency)	-0.7414
Zipf fitting R^2	-0.7690
Zipf alpha	-0.7329

5 Further Analysis of Workload Properties

We have shown the properties of the in-memory caching workloads at Twitter. In this section, we show the relationship between the properties, and how they relate to major caching use cases.

5.1 Correlations between Properties

Throughout the analysis in previous sections, we observe some workload characteristics have strong correlations with the write ratio. For example, write-heavy workloads usually use short TTLs. Presented in Figure 12a, the dashed red curve shows the mean TTL distribution of write-heavy workloads, and the solid blue curve shows the mean TTL distribution



(a) TTL distribution **(b)** Object frequency distribution **(c)** R^2 in fitting Zipfian **(d)** α of Zipfian distribution

Figure 12: Write-heavy workloads tend to show short TTL, small object access frequency, relatively large deviations from Zipfian popularity distribution and are usually less skewed (small α).

of read-heavy workloads. Around 50% of the write-heavy workloads have mean TTL shorter than 10 minutes, while for read-heavy workloads, this is 15 hours. Further, the Pearson coefficient between write ratio and log [4] of mean TTL (Table. 1) is -0.63 indicating a negative correlation, confirming that large write ratio workloads usually have short TTLs.

Besides TTL, write-heavy workloads also show low object frequencies. We present the mean object frequency (in terms of the number of accesses in the traces) of read-heavy and write-heavy workloads in Figure 12b. It shows that read-heavy workloads have a mean frequency mostly in the range from 6 to 1000, with 75% percentile above 200. Meanwhile, write-heavy workloads have a mean frequency mostly between 1 and 100, with 75% percentile below 10. We further confirm this relationship with the Pearson coefficient between write ratio and log of frequency, which is -0.7414 (Table. 1), suggesting the low object access frequency in write-heavy caches.

In addition, the popularity of write-heavy workloads has relatively larger deviations from Zipfian distribution, and the fitting confidence R^2 is usually much smaller than that of read-heavy workloads (Figure 12c). Moreover, the α parameter of Zipfian distribution in write-heavy workloads is usually small, as shown in Figure 12d. It shows the write-heavy workloads have a median α around 0.9, and the median of read-heavy workloads have an α around 1.4. This correlation is also backed up by the Pearson coefficient (Table 1).

5.2 Properties of Different Cache Use Cases

Here we further explore common properties exhibited by each of the three major caching use cases as described in Section 2.4.

5.2.1 Caching for Storage

Caches for storage usually serve ready-heavy workloads, and their popularity distributions typically follow Zipfian distribution with a large parameter α in the range of 1.2 to 2.2. While this type of workload is highly skewed, they are easier to cache, and in production, 95% of these clusters have miss ratios of around or less than 1%. Being more cacheable and having smaller miss ratios do not indicate they have small working set sizes. In our observation, 7 of the top 10 caches (ranked by cache size) belong to this category.

Because these caches store objects persisted in the backend storage, any modifications to the objects are explicitly written to both the backend and the cache. Therefore the TTLs used in these caches are usually large, in the range of days. There is no specific pattern about object size in this type of caches, and the value can be as large as tens of KB, or as small as a few bytes. For example, the number of favorites a tweet received is persisted in the backend database and sometimes cached.

5.2.2 Caching for Computation

Caches under this category serve both read-heavy and write-heavy traffic depending on the workloads. For example, machine learning feature workloads are usually read-heavy showing a good fit of Zipfian popularity distribution. While intermediate computation workloads are normally write-heavy and show deviations from Zipfian. Compared to caching for storage, workloads under this category use shorter TTLs, usually determined by the application requirement. For example, caches storing intermediate computation data usually have TTLs no more than minutes because other services will consume the data in a short time. For features and prediction results, the TTLs are usually in the range of minutes to hours (some up to days) depending on how fast the underlying data change and how expensive the computation is. The mean TTLs we observe for caches under this category is 9.6 hours. There are no particular patterns about object sizes in these caches.

Since objects stored in these caches are indirectly related to users and contents, the workloads usually have large key spaces and total working set sizes. For example, a cache storing the distance between two users will require a N^2 cache size where N denotes the number of users. However, because these caches have short TTLs, the effective working set sizes are usually much smaller. Thus removing expired objects can be more important than eviction for these caches.

As real-time stream processing becomes more popular, we envision there will be more caches being provisioned for caching computation results. Because the characteristics are different from caching for storage, they may not benefit equally from optimizations that only aim to make the

[4] We choose to use log of TTL and frequency because of their wide ranges in different workloads.

read path fast and scalable, such as optimistic cuckoo hashing [43]. Therefore, including evaluation against caching-for-computation workloads that are write-heavy and more ephemeral will paint a more complete picture of the capabilities of any caching system.

5.2.3 Transient Data with No Backing Store

There are two characteristics associated with this type of caches: Caches under this category usually have short TTLs, and the TTLs are often used to enforce implicit object deletion (Section 4.4). In addition, objects in these caches are usually tiny and we observe an average object size of 54 bytes. Although caches of this type only contribute 9% of total Twemcache cluster request rate and 8% of total cache sizes, they currently play an irreplaceable role in site operations.

6 Eviction Algorithms

We have shown the characteristics of in-memory cache workloads in the previous sections. In this section, we use the same cache traces to investigate the impact of eviction algorithms. This evaluation considers production algorithms offered by Twemcache and other production systems.

6.1 Eviction algorithm candidates

Object LRU and object FIFO LRU and FIFO are the most common algorithms used in production caching systems [4, 18]. However, they cannot be applied to systems using slab-based memory management such as Twemcache without modification. Therefore, we evaluate LRU and FIFO assuming the workloads are served using a non-slab based caching system, while ignoring memory inefficiency caused by external fragmentation. As a result, we expect that the results to have a bias toward the effectiveness of LRU and FIFO compared to the three slab-based algorithms. Production results for these two algorithms might be worse than what is suggested in this section, depending on the workloads.

slabLRU and slabLRC These two algorithms are part of eviction algorithms offered in Twemcache. slabLRU and slabLRC are equivalent to LRU and FIFO but executed at a level much coarser granularity of slabs rather than a single object. Twitter employs these algorithms to alleviate the effect of slab calcification and also to reduce the size of per-object metadata.

Random slab eviction Besides slabLRU and slabLRC, Twemcache also offers Random slab eviction, which globally picks a random slab to evict. This algorithm is workload-agnostic with robust behavior, and therefore used as the default policy in production. However, it is rarely the best of all algorithms and are non-deterministic, therefore we do not include it in comparison.

Memcached-LRU Memcached adapted LRU by creating one LRU queue per slab class. We call the resulted eviction algorithm Memcached-LRU, which does not enable Memcached's slab auto-move functionality. We did, however, evaluate Memcached-LRU with slab auto-move turned on, and most of the results are somewhere between LRU and slabLRU. The rest of the paper omits this combination.

6.2 Simulation Setup

We built an open-source simulator called libCacheSim [71] to study the steady-state miss ratio of the different eviction algorithms. Specifically, we use five-day traces to warm up the caches, then use one-day traces to evaluate cache miss ratios. Each algorithm is applied against all traces, and then grouped by results.

In terms of cache sizes, our simulation always starts with 64MB of DRAM, and chooses the maximum as 2× their current memory in production. We stop increasing the size for a particular workload when all algorithms have reached the compulsory miss ratio. Note that when plotting, the size range is truncated to better present the trend.

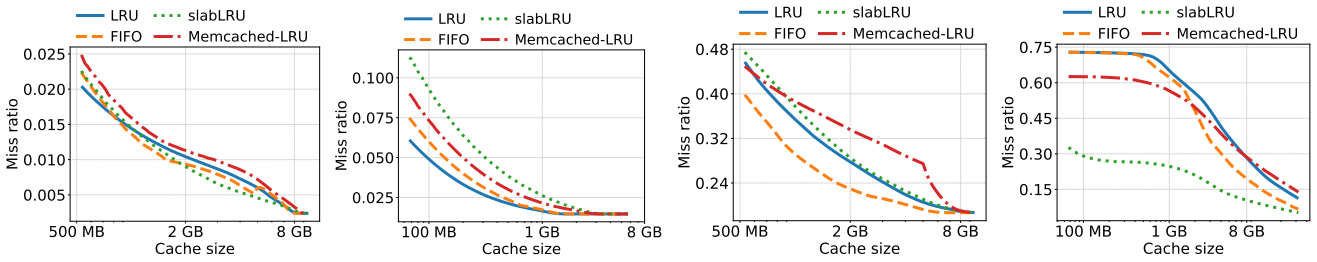
6.3 Miss Ratio Comparison

The outcome of our comparison can be grouped into four types, and representatives of each are shown in Figure 13.

The first group shows comparable miss ratios for all algorithms in the cache sizes we evaluated. For this type of workload, the choice of eviction algorithms has a limited impact on the miss ratio. Production deployments may very well favor simplicity or decide based on other operational considerations such as memory fragmentation. Twemcache uses random slab eviction by default because random eviction is simple and requires less metadata.

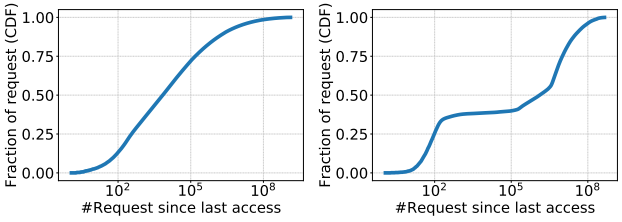
The second type of result shows that for some workloads LRU works better than others. Such a result is often expected because LRU protects recently accessed objects and is well-known for its miss ratio performance in workloads with strong temporal locality.

The third type of result shows that FIFO is the best eviction algorithm (Figure 13c). This result is somewhat surprising since it does not conform to what is typically observed in caching of other scenarios such as CDN caching. We give our suspected reasons below. Figure 14 shows the inter-arrival time distribution of the two workloads in Figure 13b and Figure 13c respectively. The inter-arrival time is the number of requests between two accesses to the same object. Figure 14a shows a smooth inter-arrival time curve, while Figure 14b shows a curve with multiple segments. For workloads with inter-arrival time like Figure 14a, LRU can work better than FIFO because it promotes recently accessed objects, which have a higher chance of being reused soon. This promotion protects the recently accessed objects but demotes other objects that are not reused recently. Demoting non-recently used objects can be an unwise decision if some of the demoted objects will be reused after 10^6 requests, such as the ones shown in Figure 14b. In contrast, FIFO treats each stored object equally; in other words, it protects the objects with a large inter-arrival gap. Therefore, for workloads similar to the one in Figure 14b, FIFO can perform better than LRU. Such



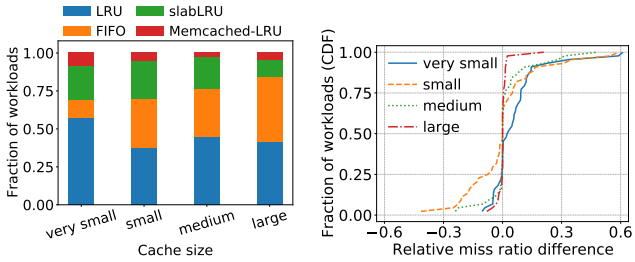
(a) Similar miss ratio **(b)** LRU is better **(c)** FIFO is better **(d)** slabLRU is better

Figure 13: Four typical miss ratio results: a) all algorithms have similar performance, b) LRU is slightly better than others, c) FIFO is better than others, d) slabLRU is much better than others.



(a) **(b)**

Figure 14: The inter-arrival gap distribution corresponding to the workloads in Figure 13b and Figure 13c respectively.



(a) **(b)**

Figure 15: a) The best eviction algorithms under different sizes. b) The relative miss ratio difference between FIFO and LRU under different sizes. Positive region shows FIFO is worse.

workloads may include scan type of requests such as a service that periodically sends emails.

The last type of result show that in some workloads, slabLRU performs much better than any other algorithms. The main reason is that the workloads showing this type of result have periodic/diurnal changes. Figure 11b shows the object size distribution over time of the workload corresponding to Figure 13d. We suspect this is due to the following reason, but we leave the verification as future work. Although LRU and FIFO are not affected by any change in object size distribution, they cannot respond to workload change instantly. In contrast, slabLRU can quickly adapt to a new workload when the new workload uses a different slab class because it prioritizes the slabs that have more recent access. From another view, slabLRU gives a larger usable cache size for the new workloads (slab class). Figure 13d shows that the difference between algorithms reduces at larger cache sizes, this is because the benefit of having a large usable cache size diminishes as cache size increases. Moreover, in these workloads, Memcached-LRU sometimes has better performance than LRU, but for most of the workloads, Memcached-LRU is worse (not shown in the figure) because of the missing capability of moving slabs. Thus it has a smaller usable cache size. When Memcached-LRU has better performance at small cache sizes, we suspect that the changing workloads cause thrashing for LRU and FIFO [27]. Since Memcached-LRU can only evict objects within from the same slab class as the new object, it protects the objects in other slab classes from thrashing, thus showing better performance.

In most cases, both miss ratio and the difference between algorithms decrease as cache capacity increases. We observe that within our simulation configuration, which stops at or before 2× current size, the difference between algorithms eventually disappears. This suggests that to achieve low miss ratio in real life, it can be quite effective to create implementations that increase the effective cache capacity, such as through metadata reduction, adopting higher capacity media, or data compression.

Given there are more than a couple of workloads showing each of the four result types, we would like to explore whether there is one algorithm that is often the best or close to the best most of the time.

In the next section, we explore how often each algorithm is the best with a special focus on LRU and FIFO.

6.4 Aggregated Statistics

In this section, we evaluate the same set of algorithms as in Section 6.3, focusing on four distinct cache sizes and present the aggregated statistics. Because different workloads have different working set sizes and compulsory miss ratios, we choose the four cache sizes in the following way. We define the *ultimate cache size* s_u to be the size where LRU achieves compulsory miss ratio for a workload. However, if LRU can not achieve compulsory miss ratio at 2× production cache size, we use 2× production cache size as s_u. We choose *large* cache size to be 90% of s_u, and *medium*, *small* and *very small*

cache sizes to be 60%, 20% and 5% of s_u respectively. We remark that, at Twitter, 76% of the caches have cache sizes larger than the *large* cache size category, and 34% of the rest have cache sizes within 10% of the *large* cache size.

We show the miss ratio comparison in Figure 15a, where each bar shows the fraction of workloads for which a particular algorithm is the best. We see that at the *large* cache size slabLRU is the best for around 10% of workloads, and this fraction gradually increases as we reduce cache size. This increase is because for smaller cache sizes, quickly adapting to workload change is more valuable. Besides this, FIFO has similar performance compared to LRU at *small, medium and large* size categories. And only at *very small* cache sizes, LRU becomes significantly better than FIFO. This is because at relatively large cache sizes, promoting recently accessed objects is less crucial. Instead, not demoting other objects is more helpful in improving the miss ratio, especially for workloads having multiple segments in inter-arrival time like the one shown in Figure 14b.

Figure 15a suggests that for close to half of the workloads, FIFO is as good as LRU at reasonably large cache sizes. Now we explore the magnitude by which FIFO is better or worse compared to LRU on each workload. Figure 15b shows the relative miss ratio difference between FIFO and LRU: $\left(\frac{mr_{\text{FIFO}} - mr_{\text{LRU}}}{mr_{\text{LRU}}}\right)$, where *mr* stands for miss ratio, for each workload at different cache sizes. When the value on X-axis is positive, it indicates that FIFO has a higher miss ratio, and LRU has better performance, while a negative value indicates the opposite. We observe that all the curves except the one for very small cache size are all close to being symmetric around x-axis value 0. This indicates that across workloads, FIFO and LRU have similar performance for small, medium and large cache sizes. For the very small size category, we observe LRU being significantly better than FIFO, this is because for workloads with temporal locality, promoting recently accessed objects becomes crucial at very small cache sizes. In production, most of the caches are running at cache sizes larger than or close to the *large* category. We believe that for most in-memory caching workloads, FIFO and LRU have a similar performance at reasonably large cache sizes.

The fact that FIFO and LRU often exhibit similar performance in production-like settings is important because using LRU usually incurs extra computational and memory overhead compared to FIFO [55, 56]. For example, implementing LRU in Memcached requires extra metadata and locks, some of which can be removed if FIFO is used.

7 Implications

In this section, we show how our observations differ from previous work, and what the takeaways are for informing future in-memory caching research.

7.1 Write-heavy Caches

Although 70% of the top twenty Twemcache clusters serve read-heavy workloads (Section 4.3.2), write-heavy workloads are also common for in-memory caching. This is not unique to Twitter. Previous work [24] from Facebook also pointed out the existence of write-heavy workloads, although the prevalence of them were not discussed due to the limited number of workloads. Furthermore, write-heavy workloads are expected to increase in prominence as the use case of caching for computation increases (Section 2.4.2). However, most of the existing systems, optimizations and research assume a read-heavy workload.

Write-heavy workloads in caching systems usually have lower throughput and higher latency, because the write path usually involves more work and can trigger more expensive events such as eviction. In Twitter's production, we observe that serving write-heavy workloads tend to have higher tail latencies. Scaling writes with many threads tends to be more challenging as well. In addition, as discussed in Section 5, write-heavy workloads have shorter TTLs with less skewed popularity, which are in sharp contrast to read-heavy workloads. This calls for future research on designing systems and solutions that consider performance on write-heavy workloads.

7.2 Short TTLs

In Section 4.4.1, we show that in-memory caching workloads frequently use short TTLs, and the usage of short TTLs reduces the effective working set size. Therefore, removing expired objects from the cache is far more important than evictions in some cases. In this section, we show that existing techniques for proactively removing expired objects (termed proactive expiration) are not sufficient. This calls for future work on better proactive expiration designs for in-memory caching systems.

Transient object cache An approach employed for proactive expiration (especially for handling short TTLs), proposed in the context of in-memory caches at Facebook [59], is to use a separate memory pool (called transient object pool) to store short-lived objects. The transient object cache consists of a circular buffer of size t with the element at index i being a linked list storing objects expiring after i seconds. Every second, all objects in the first linked list expire and are removed from the cache, then all other linked lists advance by one.

This approach is effective only when the cache user uses a mix of very short and long TTLs with the short TTL usually in the range of seconds. Since objects in the transient pool are never evicted before expiration, the size of transient pool can grow unbounded and cause objects in the normal pool to be evicted . In addition, the TTL threshold of admitting into transient object pool is non-trivial to optimize.

As we show in Figure 6b, 20% of the Twemcache workloads use a single TTL. For these workloads, transient object pool does not apply. For the workloads using multiple TTLs, we observe that fewer than 35% have their smallest TTL shorter than 300 seconds, and over 25% of caches have the smallest TTL longer than 6 hours (Figure 6c). This indicates

that the idea of transient object cache is not applicable to a large fraction of Twemcache clusters.

Background crawler Another approach for proactive expiration, which is employed in Memcached, is to use a background crawler that proactively removes expired objects by scanning all stored objects.

Using a background crawler is effective when TTLs used in the cache do not have a broad range. While scanning is effective, it is not efficient. If the cache scans all the objects every T_{pass}, an object of TTL t can be scanned up to $1+\lceil\frac{t}{T_{pass}}\rceil$ times before removal, and can overstay in the system by up to T_{pass}. The cache operator therefore has to make a tradeoff between wasted space and the additional CPU cycles and memory bandwidth needed for scanning. This tradeoff gets harder if a cache has a wide TTL range, which is common as observed in Section 4.4. While the Twemcache workloads are single tenant, wide TTL range issue would be further exacerbated for multi-tenant caches.

Figure 6d shows that TTLs used within each workload have a wide range. Close to 60% of workloads have the maximum TTL more than twice as long as the minimum, and 25% of workloads show a ratio at or above 100. This indicates that for the 25% of caches, if we want to ensure all objects are removed within 2× their TTLs, objects with the longest TTL will be scanned 100 times before expiration.

The combination of transient object cache with background crawler could extend the coverage of workloads that can be efficiently expired. However, the tradeoff between wasted space and the additional CPU cycles and memory bandwidth consumed for scanning would still remain. Hence, future innovation is necessary to fundamentally address use cases where TTLs exhibit a broad range.

7.3 Highly Skewed Object Popularity

Our work shows that the object popularity of in-memory caching can be far more skewed than previously shown [19], or compared to studies on web proxy workloads [29] and CDN workloads [47]. We suspect this has a lot to do with the nature of Twitter's product, which puts great emphasis on the timeliness of its content. It remains to be seen whether this is a widespread pattern or trend. Cache workloads are also more skewed compared to NoSQL database such as RocksDB [38], which is not surprising because database traffic is often already filtered by caches, and has the most skewed portion removed via cache hits. In other words, in-memory caching and NoSQL database often observe different traffic even for the same application. Besides these two reasons, sampling sometimes results in bias in the popularity modelling, and we avoid this by collecting unsampled traces. Our observation that the workloads still follow Zipfian distribution with large alpha value emphasizes the importance of addressing load imbalance [44, 57, 61].

7.4 Object Size

Similar to previously reported [24], we observe that objects cached in in-memory caching are often tiny (Section 4.6). As a result, in-memory caches are not always bound by memory size; instead, close to 20% of the Twemcache clusters are CPU-bound.

On the other hand, small objects signifies the relative large overhead of metadata. Memcached stores 56-byte with each object, and Twitter's current production cache uses 38-byte metadata with each object. Reducing object metadata further can yield substantial benefits for caching tiny objects.

In addition, we observe that compared to value size, the key size can be large in some workloads. For 60% of the workloads, the mean key size and mean value size are in the same order of magnitude. This indicates that reducing key size can be very important for these workloads. Many workloads we observed have namespaces as part of the object keys, such as `NS1:NS2:...:id`. This format is commonly used to mirror the naming in a multi-tenant database, which is also observed at Facebook [32]. Namespaces thus can occupy large fractions of precious cache space while being highly repetitive within a single cache cluster. However, there is no known techniques to "compress" the keys. To encourage and facilitate future research on this, we keep the original but anonymized namespace in our open sourced traces.

Several recent works [26, 28] on reducing miss ratio (improving memory efficiency) focused on improving eviction algorithms and often add more metadata. Given our observations here, we would like to call more attention to the optimization of cache metadata and object keys.

7.5 Dynamic Object Size Distribution

In Section 4.6.2, we show that the object size distribution is not static, and the distribution shifts over time can cause out-of-memory (OOM) exceptions for caching systems using external allocators, or slab calcification for those using slab-based memory management. In order to solve this problem, one solution, employed by Facebook, is to migrate slabs between slab classes by balancing the age of the oldest items in each class [59]. Earlier versions of Memcached approached this problem by balancing the eviction rate of each slab class. Since version 1.6.6, Memcached has also moved to using the solution of balancing the age as mentioned above.

Besides efforts in production systems, slab assignment and migration has also been a hot topic in recent research [31, 35, 36, 46]. However, to the best of our knowledge, the problem has only been studied under a "semi-static" request sequence. Specifically, the research so far assumes that the miss ratio curve or some other properties of each slab class hold steady for =certain amount of time, which often precludes periodic and sudden changes in object size distribution.

In general, the temporal properties of object sizes in cache are not well understood or quantified. As presented in Figure 11c and Figure 11d, it is not rare to see unexpected

changes in size distribution only lasting for a few hours. Sometimes it is hard to pinpoint the root cause of such changes. Nonetheless, we believe that temporal changes related to object size, whether recurring or as a one-off, usually have drivers with roots beyond the time dimension. For example, the tweet size drift throughout the day may very well depend on the locales or geo-location of active users. Some caches may be shared by datasets which differ in size distribution and access cycles, resulting in different distributions dominating the access pattern at different instants of the day. In this sense, studying the object size distribution over time could very well provide deeper insights into characteristics of the datasets being cached. Considering the increasing interest in using machine learning and other statistical tools to study and predict caching behavior, we think object size dynamics might provide a good proxy to evaluate the relationship between basic dataset attributes and their behavior in cache, allowing caching systems to make smarter decisions over time.

8 Related Work

Due to the nature of this work, we have discussed related works in detail throughout the paper.

Multiple caching and storage system traces were collected and analyzed in the past [24, 25, 32, 47, 48, 59]; however, only a limited number of reports focus on in-memory caching workloads [24,48,59]. The closet work to our analysis is Facebook's Memcached workload analysis [24], which examined five Memcached pools at Facebook. Similar to the observations in this work [24], we observe the sizes of objects stored in Twemcache are small, and diurnal patterns are common in multiple characteristics. After analyzing 153 Twemcache clusters at Twitter, in addition to previous observations [24], we show that write-heavy workloads are popular. Moreover, we focus on several aspects of in-memory caching which have not been studied to the best of our knowledge , including TTL and cache dynamics. Although previous work [24] proposed analytical models on the key size, value size, and inter-arrival gap distribution, the models do not fully capture all the dimensions of production caching workloads such as changing working set and dynamic object size distribution. Compared to synthetic workload models, the collection of real-world traces that we collected and open sourced provide a detailed picture of various aspects of the workloads of production in-memory caches. .

Besides workload analysis on Memcached, there have been several workload analysis on web proxy [21–23, 49, 64] and CDN caching [47, 67]. The photo caching and serving infrastructure at Facebook has been studied [47], with a focus on the effect of layering in caching along with the relationship between content popularity, age, and social-networking metrics.

In addition to caching in web proxies and CDNs, the effectiveness of caching is often discussed in workload studies [25,60,66] of file systems. However, these works primarily studied the cache to the extent that of its effectiveness in reducing traffic to the storage system rather than on aspects that affect the design of the cache itself. Besides, file system caching is different from distributed in-memory caches due to a variety of reasons. For example, file system caches usually stores objects of fixed-sized chunks (512 bytes, 4 KB or larger), while in-memory caches store objects of a much wider range (Section 4.6), and scan is common in file systems, while rare in in-memory caches.

Because of the similarities in the interface, in-memory caching is sometimes discussed together with key-value databases. Three different RocksDB workloads [32] at Facebook has been studied in depth, with a focus on the distribution of key and value sizes, locality, and diurnal patterns in different metrics. Although Twemcache and RocksDB have a similar key-value interface, they are fundamentally different because of their design and usage. RocksDB stores data for persistence, while Twemcache stores data to provide low latency and high throughput without persistence. In addition, compared to RocksDB, TTL and evictions are unique to in-memory caching.

9 Conclusion

We studied the workloads of 153 in-memory cache clusters at Twitter and discovered five important facts about in-memory caching. First, although read-heavy workloads account for more than half of the resource usages, write-heavy workloads are also common. Second, in-memory caching clients often use short TTLs, which limits the effective working set size. Thus, removing expired objects needs to be prioritized before evictions. Third, read-heavy in-memory caching workloads follow Zipfian popularity distribution with a large skew. Fourth, the object size distributions of most workloads are not static. Instead, it changes over time with both diurnal patterns and sudden changes, highlighting the importance of slab migration for slab-based in-memory caching systems. Last, for a significant number of workloads, FIFO has similar or lower miss ratio performance as LRU for in-memory caching workloads. We have open sourced the traces collected at https://github.com/twitter/cache-trace.

Acknowledgements We thank our shepherd Andrea Arpaci-Dusseau and the anonymous reviewers for their valuable feedback. We thank our colleagues Jack Kosaian and Rebecca Isaacs for their extensive reviews and comments that improved this work. We also want to thank the Cache team and IOP team at Twitter for their support in collecting and analyzing the traces, and Daniel Berger for his comments in the early stage of the project. Moreover, we thank CloudLab [40] in helping us process the open-sourced traces, and Geoff Kuenning from SNIA in helping hosting and sharing the traces. This work was supported in part by NSF grants CNS 1901410 and CNS 1956271.

References

[1] Anonymized twitter production cache traces. https://github.com/twitter/cache-trace.

[2] Apache aurora. http://aurora.apache.org/. Accessed: 2020-05-06.

[3] Apache mesos. http://mesos.apache.org/. Accessed: 2020-05-06.

[4] Apache traffic server. https://trafficserver.apache.org/. Accessed: 2020-05-06.

[5] Art. 17 gdpr right to erasure ('right to be forgotten'). https://gdpr-info.eu/art-17-gdpr/. Accessed: 2020-05-06.

[6] Caching with twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache.html. Accessed: 2020-10-10.

[7] database caching strategy using redis. https://d0.awsstatic.com/whitepapers/Database/database-caching-strategies-using-redis.pdf. Accessed: 2020-05-06.

[8] Decomposing twitter: Adventures in service-oriented architecture. https://www.infoq.com/presentations/twitter-soa/. Accessed: 2020-09-25.

[9] Do not join lru and slab maintainer threads if they do not exist. https://github.com/memcached/memcached/pull/686. Accessed: 2020-08-06.

[10] Enhance slab reallocation for burst of evictions. https://github.com/memcached/memcached/pull/695. Accessed: 2020-08-06.

[11] Experiencing slab ooms after one week of uptime. https://github.com/memcached/memcached/issues/689. Accessed: 2020-08-06.

[12] How to interpret r-squared and goodness-of-fit in regression analysis. https://www.datasciencecentral.com/profiles/blogs/regression-analysis-\how-do-i-interpret-r-squared-and-assess-the. Accessed: 2020-09-28.

[13] jemalloc. http://jemalloc.net/. Accessed: 2020-05-06.

[14] memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2020-05-06.

[15] Paper review: Memc3. https://memcached.org/blog/paper-review-memc3/. Accessed: 2020-05-06.

[16] Redis. http://redis.io/. Accessed: 2020-05-06.

[17] slab auto-mover anti-favours slab 2. https://github.com/memcached/memcached/issues/677. Accessed: 2020-08-06.

[18] Varnish cache. https://varnish-cache.org/. Accessed: 2020-05-06.

[19] The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.

[20] Mehmet Altinel, Christof Bornhoevd, Chandrasekaran Mohan, Mir Hamid Pirahesh, Berthold Reinwald, and Saileshwar Krishnamurthy. System and method for adaptive database caching, July 1 2008. US Patent 7,395,258.

[21] Martin Arlitt, Rich Friedrich, and Tai Jin. Workload characterization of a web proxy in a cable modem environment. *ACM SIGMETRICS Performance Evaluation Review*, 27(2):25–36, 1999.

[22] Martin Arlitt and Tai Jin. A workload characterization study of the 1998 world cup web site. *IEEE network*, 14(3):30–37, 2000.

[23] Martin F Arlitt and Carey L Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on networking*, 5(5):631–645, 1997.

[24] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[25] Mary G Baker, John H Hartman, Michael D Kupfer, Ken W Shirriff, and John K Ousterhout. Measurements of a distributed file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 198–212, 1991.

[26] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd : Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[27] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75. IEEE, 2015.

[28] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.

[29] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99*, volume 1, pages 126–134. IEEE, 1999.

[30] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[31] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*, pages 353–362, 2019.

[32] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[33] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, February 2020. USENIX Association.

[34] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[36] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[37] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[38] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[39] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, 2019.

[40] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, jul 2019.

[41] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[42] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[43] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[44] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–12, 2011.

[45] Wolfram Gloger. ptmalloc. http://www.malloc.de/en/. Accessed: 2020-05-06.

[46] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, 2015.

[47] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of

facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181, 2013.

[48] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2014.

[49] Sunghwan Ihm and Vivek S Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 295–312, 2011.

[50] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[51] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. Dns performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 153–167, 2001.

[52] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. Slik : Scalable low-latency indexes for a key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 57–70, 2016.

[53] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[54] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[55] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 476–488, 2015.

[56] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica : A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[57] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[58] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce G Lindsay, and Jeffrey F Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611, 2002.

[59] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[60] John K Ousterhout, Herve Da Costa, David Harrison, John A Kunze, Mike Kupfer, and James G Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, pages 15–24, 1985.

[61] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, Savannah, GA, November 2016. USENIX Association.

[62] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*, San Jose, CA, USA, October 2012.

[63] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, November 2011. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[64] Weisong Shi, Randy Wright, Eli Collins, and Vijay Karamcheti. Workload characterization of a personalized web site and its implications for dynamic content caching. In *Proceedings of the 7th International Workshop on Web Caching and Content Distribution (WCW'02)*. Citeseer, 2002.

[65] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq : Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[66] Werner Vogels. File system usage in windows nt 4.0. *ACM SIGOPS Operating Systems Review*, 33(5):93–109, 1999.

[67] Patrick Wendell and Michael J Freedman. Going viral: flash crowds in an open cdn. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 549–558, 2011.

[68] wikimedia. Analytics/data lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching. Accessed: 2020-05-06.

[69] Wikimedia. caching overview - wikitech. https://wikitech.wikimedia.org/wiki/Caching_overview. Accessed: 2020-05-06.

[70] John Wilkes. More Google cluster data. Google research blog, November 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[71] Juncheng Yang. libcachesim. https://github.com/1a1a11a/libCacheSim. Accessed: 2020-09-28.

[72] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 66–79, 2017.





Generalized Sub-Query Fusion for Eliminating Redundant I/O from Big-Data Queries

Partho Sarthi, Kaushik Rajan, Akash Lal
Microsoft Research India

Abhishek Modi, Prakhar Jain, Mo Liu, Ashit Gosalia
Microsoft

Saurabh Kalikar[*]
Intel

Abstract

SQL is the de-facto language for big-data analytics. Despite the cost of distributed SQL execution being dominated by disk and network I/O, we find that state-of-the-art optimizers produce plans that are not I/O optimal. For a significant fraction of queries (25% of popular benchmarks like TPCDS), a large amount of data is shuffled redundantly between different pairs of stages. The fundamental reason for this limitation is that optimizers do not have the right set of primitives to perform reasoning at the map-reduce level that can potentially identify and eliminate the redundant I/O.

This paper proposes RESIN, an optimizer extension that adds first-class support for map-reduce reasoning. RESIN uses a novel technique called *Generalized Sub-Query Fusion* that identifies sub-queries computing on overlapping data, and *fuses* them into the same map-reduce stages. The analysis is general; it does not require that the sub-queries be syntactically the same, nor are they required to produce the same output. Sub-query fusion allows RESIN to sometimes also eliminate expensive binary operations like *Joins* and *Unions* altogether for further gains.

We have integrated RESIN into SPARKSQL and evaluated it on TPCDS, a standard analytics benchmark suite. Our results demonstrate that the proposed optimizations apply to 40% of the queries and speed up a large fraction of them by $1.1-6\times$, reducing the overall execution time of the benchmark suite by 12%.

1 Introduction

SQL is the de-facto language for performing big-data analytics. As there are many alternative ways to express the same query in SQL, query optimizers employ SQL-to-SQL rewrite rules to find equivalent queries that are likely to run faster. The rewritten query is compiled down to an executable plan that consists of many map or reduce stages. Each stage in the plan then runs in a data-parallel manner on many machines. Data is materialized to disk at the end of each stage and transferred between stages using an all-to-all network *shuffle* (also referred to as an *exchange*). In practice, shuffles that involve very large amount of data require multiple rounds of I/O in order to incrementally aggregate data [15,27]. It is not surprising therefore, that the cost of running a query is dominated by disk and network I/O [15].

Despite the bottleneck on I/O, we find that state-of-the-art query optimizers [4, 5, 18, 21, 22] produce execution plans that read or shuffle the same data redundantly multiple times. In fact, on a standard benchmark like TPCDS, the SPARK query optimizer produces plans where 40% of queries incur redundant I/O (Section 6). A large fraction of these spend at-least half their time in stages with redundant I/O.

A standard big-data query optimizer (Figure 1) performs query optimization using a sequence of tree-rewrite rules. It applies *logical* rules to substitute operator trees with equivalent trees. Then it uses implementation strategies (also called *physical* rules) to transform an optimized operator tree into a tree of physical operators. Each physical operator has a predefined map-reduce implementation. As shown in the figure, a standard optimizer only performs SQL-to-SQL rewrite rules at the logical level and the physical operators just provide data-parallel implementations of SQL operators.

Performing optimization at the SQL level is not optimal for a runtime that can execute arbitrary data-parallel operators. A previous system called BLITZ [10, 19] shows evidence of this opportunity for further optimization. BLITZ [19] uses program synthesis to identify single-input single-output sub-queries that can be implemented by a single imperative map-reduce program. Through program synthesis, it finds map-reduce implementations of queries where a query optimizer produces inefficient execution plans. Subsequent work [10] added some of the newly discovered operators (referred to as *super-operators*) back into the optimizer along with rewrite-rules that target them. They showed that queries that use a super-operator can run up to $2\times$ faster.

A key limitation of BLITZ, however, is that it only optimizes single-input sub-queries, and further it only targets optimiza-

[*]Work was done while the author was at Microsoft

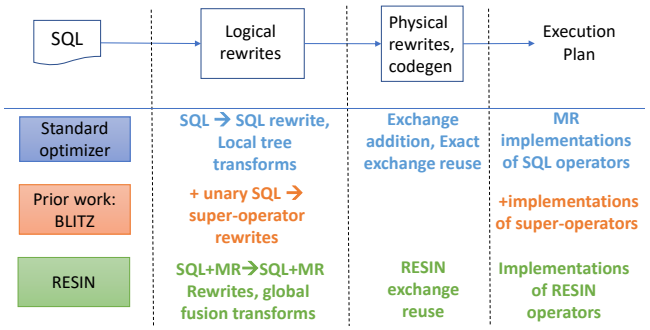


Figure 1: Key components of a big-data query optimizer. RESIN performs map-reduce reasoning starting at the logical level, when prior work only considers it late in the process.

tions that produce a single map-reduce program. The logical rewrite rules in BLITZ (as in a standard optimizer) only make local transformations. They substitute a connected set of operators with a single super-operator. As a result, BLITZ can only eliminate redundant I/O from specific types of sub-queries with *self-joins* or *self-unions*. This turned out to be insufficient on a standard benchmark suite like TPCDS where BLITZ only applies to a small fraction (2%) of queries.

This paper introduces RESIN, an optimizer extension that eliminates redundant I/O from complex multi-stage multi-input queries. This fundamentally requires new techniques. As shown in Figure 1, RESIN performs map-reduce reasoning right from the beginning. It introduces two generic logical operators, a parameterized *mapper* (RESINMAP) and a parameterized *reducer* (RESINREDUCE) that are each capable of implementing complex sub-queries. RESIN introduces new rules that *fuse* operators from different parts of the query tree that are processing overlapping sets of data. The fusion relies on the additional expressiveness of RESINMAP and RESINREDUCE. The fusion further enables the elimination of binary operators from the query. Binary operators are particularly expensive as they typically induce multiple shuffles [10]. Compared to BLITZ, we significantly broaden the optimization opportunities: RESIN applies to 38% of TPCDS.

We integrated RESIN with SPARK [5, 26], a popular open-source big-data system, and evaluated on the entire TPCDS suite. Our results demonstrate that RESIN optimizations apply to 40 of the 104 queries in the suite, and speed up 25% of the queries by a significant fraction (average 1.4×). RESIN brings down the cumulative execution time for the entire benchmark suite by 12%.

The rest of the paper is organized as follows. Section 2 gives an overview of optimizations performed by RESIN. Section 3 formally defines the query language and introduces RESIN operators. Section 4 describes the core optimizations, sub-query fusion and binary operator elimination. Section 5 presents some key features of our implementation. Section 6 reports our evaluation and Section 7 discusses related work.

2 Overview

This section provides an overview of RESIN. Consider a (fictitious) IoT application that collects readings from multiple sensors deployed all over the world and derives intelligence from it through SQL queries. Each device emits a single message every few hours with two readings corresponding to two different times. The message has the following fields, $\langle id, hr_1, signal_1, hr_2, signal_2 \rangle$ where *id* is the device identifier, hr_1 is the hour at which the first reading was taken, $signal_1$ is the value of the first reading. Similarly, hr_2 and $signal_2$ are the hour and value of the second reading. The collective log, which can reach Billions of entries across all devices, is processed once a month using SQL queries. We describe RESIN optimizations on two example queries.

Example 1 The query is shown in Figure 2(a)[1]. The query separates the subset of columns $\langle id, hr_1, signal_1 \rangle$ and $\langle id, hr_2, signal_2 \rangle$ of each row of the *rawLogs* table to get intermediate tables $V1$ and $V2$, and then performs a *Union* to put them together. The *Union* operator performs a multi-set union, i.e., it does not remove duplicate rows from the output. (In general, all our queries operate with multi-set semantics.) Each of $V1$ and $V2$ additionally requires a filter to check for the validity of the input (*hr* fields are in the expected ranges and the *signal* fields are valid). Figure 2(b) shows a small input table with 5 rows and the result of executing the query on that input. Each of $V1$ and $V2$ will contain 4 rows each and the final output *signals* has 8 rows. For a production-sized execution, imagine scaling each table by a factor of a Billion.

Figure 2(c) shows the execution plan for this query generated by SPARK. The plan employs duplicate scan operators, thus, it reads the same input twice. Even if there is an index on the input (in fact, we are going to assume a perfect index that can filter out irrelevant rows), many rows (R_2, R_3, R_5) would still be read twice (because they are needed for both $V1$ and $V2$) and processed independently. Unfortunately, SQL's relational operators provide no better way of expressing the query because there is no way to produce multiple output rows for each input row, other than by using a *Union* operator as in this example. When the inputs to the *Union* have a common source, the binary operator induces redundant I/O. This example shows a case where input data is read redundantly, however in general a *Union* could induce redundant shuffles as well.

There is a better way to implement the query directly using map-reduce operators. Consider the mapper shown in Figure 3. It reads and processes each input row once, producing up to two output rows per input row. The mapper applies the filters (Line 4 and Line 7) one after the other and outputs the relevant columns. (We operate in the standard *multi-set*

[1]We show queries as a sequence of statements for the ease of illustration. They could have instead be written as a single nested query; our optimizations still apply in the same manner.

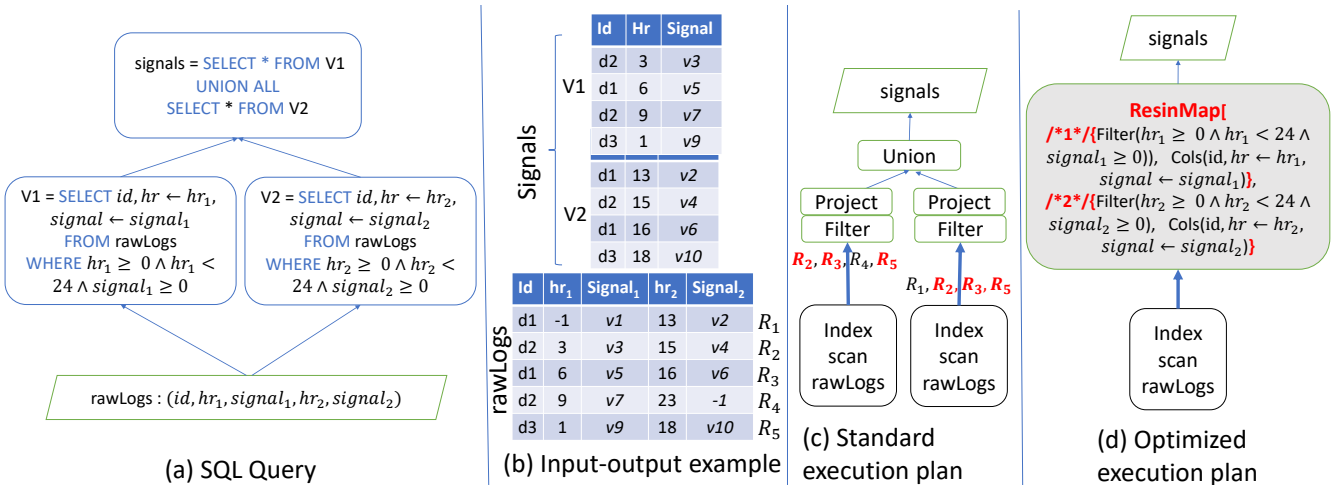


Figure 2: A SQL query, example input-outputs, execution plan showing redundant I/O and an optimized plan produced by RESIN.

```
1  //Mapper m processes a partition rawlogs[m]
2  method exampleResinMap(m){
3    foreach ⟨id,hr1,signal1,hr2,signal2⟩ ∈ rawLogs[m] {
4      if (hr1 ≥ 0∧hr1 < 24∧signal1 ≥ 0) {
5        hr = hr1; signal = signal1; output(id,hr,signal);
6      }
7      if (hr2 ≥ 0∧hr2 < 24∧signal2 ≥ 0) {
8        hr = hr2; signal = signal2; output(id,hr,signal);
9  } } }
```

Figure 3: A mapper that implements the query Figure 2.

semantics of SQL, so the order of rows in an output table is immaterial.) The mapper is sufficient to implement our example query. The reason current optimizers do not consider this option is that they do not reason at the level of mappers (or reducers) during optimization. They only reason about SQL operators and perform SQL-to-SQL query rewrites.

RESIN extends the optimizer with a generic map operator RESINMAP and a generic reduce operator RESINREDUCE (used in the next example). RESINMAP is a row-wise operator that may produce zero or more output rows for each input row. The generated plan with RESIN for our example query is shown in Figure 2(d). The plan uses a RESINMAP operator. (The code generated for this operator is essentially the one in Figure 3.) RESINMAP consists of multiple entries (two in the example, marked 1 and 2 in the Figure), each with a filter and associated expressions to produce output. The RESINMAP operator is quite powerful. It can implement any single-input single-output sub-query containing arbitrary combinations of *Select*, *Project* and *Union* operators.

Example 2 As a second example, consider the more complex query shown in Figure 4. The query has two inputs, the *signals* table, which comes from the output of the previous example, and another table called *dInfo* that has device-specific

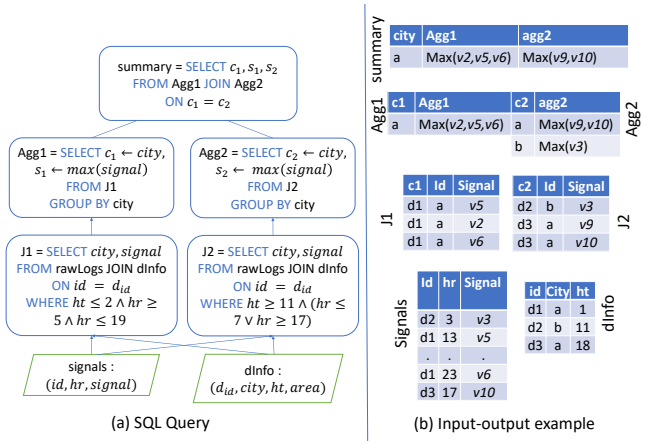


Figure 4: A SQL query with an input-output example.

information. Each row of *dInfo* contains a device identifier d_{id}, the *city* of deployment of the device, and the height *ht* at which the device is installed. The query works as follows. Its result is the *Join* of two intermediate tables $Agg1$ and $Agg2$. The table $Agg1$ contains the maximum day-time reading ($5 \leq hr \leq 19$) per city among all devices deployed at ground level ($ht \leq 2$). This itself requires a join on the *signals* and *dInfo* tables. The table $Agg2$ similarly is the maximum night-time ($hr \geq 17 \vee hr \leq 7$) reading for devices deployed at a height above the ground level ($ht \geq 11$).

A *Join* operator is parameterized by a predicate in the *ON* clause. It takes all combinations of pairs of rows from its input tables, concatenates them and filters according to the *ON* condition. A common usage of *Join* is an *Equi-Join*, where the *ON* clause equates the values of one (or more) columns in the first argument table with one (or more) columns in the second argument table. The query contains three equi-joins, two on the device identifier (for $J1$ and $J2$) and one on the

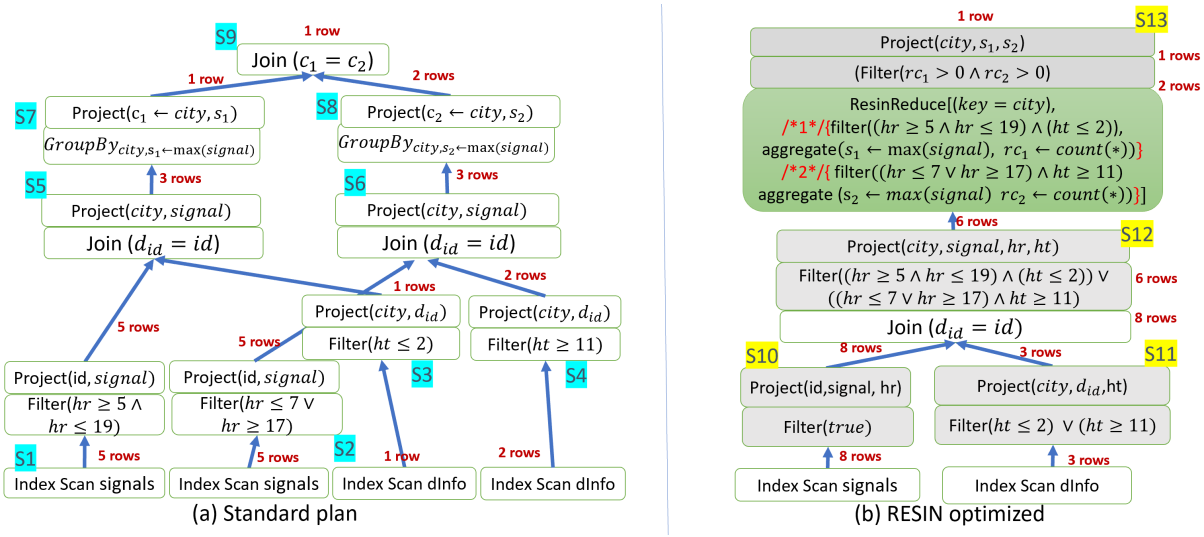


Figure 5: Query execution plans before and after RESIN optimization.

city (for the final output *summary*).

Aggregation happens via the *GroupBy* operator; it partitions its input table on unique values of the grouping key and performs an aggregation on each partition. It necessarily produces only one output row per partition. For instance, the computation of $Agg1$ partitions the table $J1$ on $city$, and for each partition it computes the maximum *signal* value using the aggregation *max*.

Figure 5 shows the execution plan generated by SPARK for this query (solid lines indicate I/O). As before, there is an index scan used each time the query references an input table. The implementation of an equi-join requires its inputs to be partitioned on equated columns, so shuffles are introduced along both arguments for all of the joins. Standard predicate push-down rules in a query optimizer will push the filters on *hr* and *ht* below join and into the respective scans in order to reduce the amount of data shuffled. Similarly, a *GroupBy* requires that its inputs be partitioned by the grouping key. So a shuffle is introduced before each *GroupBy*. In total, the plan has 9 stages; four for scanning input tables ($S1$, $S2$, $S3$, $S4$), three for the *Join* operators ($S5$, $S6$, $S9$), and a further two for the *GroupBy* operators ($S7$, $S8$).

This plan has many sources of redundant I/O. First, some rows of the *signals* table (e.g., with signal values $v5$ and $v10$) are redundantly scanned. Note that the redundant scan happens despite having the best possible indices because some rows can satisfy both the filters on *hr*. The scanned tables are then partitioned on the same column (*id*) and shuffled to the respective join operators ($J1$ and $J2$). A shuffle is a partitioning operator, it takes as input a partitioning key and a partition count, and partitions the input rows according to the key. We say two shuffle operators redundantly shuffle a row if (a) the key column for the two shuffles has the same value, and (b) all the columns of the row are derived from a common set of source tables. For our example, the two rows corresponding to $v5$ and $v10$, are redundantly shuffled before the join because they come from the same input row in *signals* table. Furthermore, as the left and right aggregates are computed separately, the aggregated results for the same city (city *a* for our example) are computed and shuffled redundantly.

Figure 5(b) is the optimized plan generated by RESIN. It has only 4 stages, each table is scanned once and no redundant shuffles. On a real dataset, a query with this structure (TPCDS $Q90$ for example) would speedup by $2\times$.

RESIN eliminates redundant I/O through two key techniques: *sub-query fusion* and *binary operator elimination*. Sub-query fusion merges operators from different parts of the query if they process the same data. More formally, given two sub-queries $Q1$ and $Q2$, the fusion rule attempts to construct a triple $\langle Q, ResinMap_1, ResinMap_2 \rangle$ such that $Q_1 = ResinMap_1(Q)$ and $Q_2 = ResinMap_2(Q)$.

For our example query, RESIN first merges the filters and projects applied on each of the input tables. $S1$ and $S2$ is merged into a single RESINMAP operator to obtain $S11$[2]. Similarly, $S3$ and $S4$ are merged into a single RESINMAP operator $S12$. Notice that the filters are combined with a disjunction and projected columns are unioned, so that all of the data required by the query is read in one go.

The fusion process then recursively moves up the tree and the two joins ($J1$ and $J2$) are merged together into a single join (J) that computes both the results. An additional RESINMAP is added right after to ensure that only rows required by either $J1$ or $J2$ are retained. *A salient feature of the fusions rules is that they ensure that the computation of the fused query Q does not shuffle more rows than the individual queries.*

[2]All the light shaded *Filter, Project* chains in Figure 5 actually represent RESINMAP. We do not show the RESINMAP explicitly, as we did in Figure 2(d), for ease of exposition.

```
1  // ps ∈ partition(J, [city,signal,ht,hr]);
2  max1 = max2 = ∞;
3  rc1 = rc2 = 0;
4  foreach (⟨city,signal,ht⟩ in ps.group) {
5    if (hr ≤ 19 ∧ hr ≥ 5 ∧ ht ≤ 2 {
6      max1=max(max1,signal); rc1 = rc1 + 1;
7    }
8    if ((hr ≥ 17 ∨ hr ≤ 7) ∧ ht ≥ 11) {
9      max2=max(max2,signal); rc2 = rc2 + 1;
10   }
11 }
12 output (city, max1, max2, rc1,rc2);
```

Figure 6: A reducer for the query plan in Figure 5.

Note once again that the output of the individual joins $J1$ and $J2$ can be separated out by applying appropriate filters on J. The reader can verify that $J1 = Select(hr \leq 19 \wedge hr \geq 5 \wedge ht \leq 2)$ *from* J and $J2 = Select((hr \geq 17 \vee hr \leq 7) \wedge ht \geq 11)$ *from* J.

Finally, the two aggregations are fused together. Note that the aggregates need to be applied on different filtered subsets of J. It turns out that fusion of aggregation operations cannot be done with standard SQL operators, neither with RESINMAP alone. RESIN *introduces a generic reduce operator* RESINREDUCE *that makes the optimization process much more expressive by directly considering map-reduce plans.* As seen in Figure 5, the two *GroupBy* are fused into a RESINREDUCE operator. The RESINREDUCE operator is parameterized by a (partition) key (in this case, *city*) and a list with two entries. Each entry contains the filter that determines a subset of rows to be aggregated as well as the aggregation function. In addition, each entry has a $count(*)$ aggregation for reasons described below.

Figure 6 shows the code that implements the reducer, applied to each partition of J (partitioned on *city*) independently. The reducer maintains variables max_1, max_2, rc_1, rc_2 for computing four aggregations. It then iterates over the partition, and for each row, it updates the aggregation variable if the row satisfies the corresponding predicate. The reducer outputs one row per partition, with five columns: the grouping key *city* and the four aggregated values.

The variables rc_1 and rc_2 are used to check if an aggregation was even applied for a partition. This is necessary to obtain back the output of the original *GroupBy*s. For our example, the output of the reducer[3] will contain 2 rows corresponding to cities a and b, however for $city = b$, $rc_1 = 0$, indicating that $Agg1$ has no output for $city = b$.

Once the aggregations are fused, RESIN performs *binary operator elimination* to get rid of the final *Join* ($S7$) altogether. RESIN figures out that the join was doing nothing more than putting together the aggregates from the two sub-queries, which is already done in the output of the RESINREDUCE operator. RESIN replaces the *Join* with a simple filter that ensures that a row is output only if both aggregates produce an output, as is dictated by the semantics of a join.

In summary, RESIN introduces a class of optimizations that target map-reduce operators to eliminate redundant I/O.

[3] We have not shown the output of the fused query; its output is the union of the original fused queries (with extra columns).

3 Preliminaries

We use a query language based on SPARKSQL [5] to present our analysis formally. We define a table as a multi-set of *rows* that each follow the same *schema*. A schema S is a set of pairs of column name and data type: $\{\langle a_1,t_1\rangle\langle a_2,t_2\rangle\cdots,\langle a_n,t_n\rangle\}$. A row r that follows schema S is a tuple of form $\{a_1 : v_1,\cdots,a_n : v_n\}$ that assigns a value v_i of type t_i to column a_i of the schema. In this case, we say $r.a_i = v_i$. We will not explicitly refer to the data-types of columns in the rest of this paper because it is not relevant to our analysis.

3.1 SQL Operators

This section defines the core SQL operators of our query language. We assume a generic syntax for *expressions* that can be evaluated over a row to produce a scalar data value. A *predicate* is simply an expression that evaluates to a Boolean value. Our implementation supports all SPARKSQL expressions and predicates.

Select $T_2 = \sigma[\phi](T_1)$

A *Select* operator discards rows of T_1 that do not satisfy the filter predicate ϕ.

Project $T_2 = \pi[map(c_i \leftarrow e_i)](T_1)$

A *Project* is parameterized by a map of $\langle c_i, e_i\rangle$ pairs, where c_i is a column name and e_i are expressions. *Project* is a row-wise operator. It iterates over all the rows of the input table T_1 and for each row, it applies the expressions e_i to compute data values of output columns c_i. Note that *Project* can be used to create *aliases* of existing columns. For instance, the operator $\pi[c_{new} \leftarrow c_{old}]$ renames input column c_{old} to the output column c_{new}. We sometimes write this operator as $\pi[C \leftarrow E]$ where C is a list of column names and E is a list of expressions and $\|C\| = \|E\|$.

GroupBy $T_2 = \gamma[K, map(c_i \leftarrow agg_i(col_i))](T_1)$

A *GroupBy* partitions the input table T_1 by unique values of columns K and applies aggregations agg_i over each partition. A partition is also referred to as a *group*. Each aggregation agg_i applies a commutative and associative function (e.g., sum, min, max, etc.) over a single column col_i of T_1. Each column of the output table is either the result of an aggregation or a key column. We sometimes write a *GroupBy* as $\gamma[K, C \leftarrow A(Col)]$ where C and Col are lists of column names, A is a list of aggregations, and $\|C\| = \|A\| = \|Col\|$.

```
1  method ResinMapOperator(T, μ[L]) {
2    foreach(row in T) {
3      foreach(⟨ϕ,C ← E⟩ in L) {
4        if(ϕ(row)){
5          for(i in 1..|E|) out.C[i] ← E[i](row)
6          output(out)
7        }
8      }
9    }
10 }
```

Figure 7: RESINMAP Operator

```
1  method ResinReduceOperator(G, ρ[K,L]) {
2    foreach (⟨ϕ_i,c_i,agg_i(col_i)⟩ in L)
3      out.c_i ← init(agg_i)
4    foreach(row in G.rows) {
5      foreach (⟨ϕ_i,c_i,agg_i(col_i)⟩ in L)
6        if (ϕ_i)  out.c_i = agg(out.c_i,row.col_i)
7    }
8    output(G.keys,out)
9  }
```

Figure 8: RESINREDUCE Operator

Join (equi-join) $T_2 = \bowtie[\psi, jt](T_{left}, T_{right})$

A *Join* is a binary operator that matches rows from T_{left} with rows from T_{right} on a conjunction of equality predicates ψ of the form $(a_1 = b_1 \wedge a_2 = b_2 \ldots \wedge a_n = b_n)$, where a_i are columns of T_{left} and b_i are columns of T_{right}. The operator requires that the columns names of the two input arguments be distinct. Parameter *jt* is a *join type* and can be any of *inner* (i), *leftOuter* (lo), *rightOuter* (ro), *leftSemi* (ls), or *rightSemi* (rs) with the standard semantics [5]. For simplicity we only define rules for inner joins in this paper. Our implementation handles other types as well.

Union $T_2 = \uplus(T_{left}, T_{right})$

A *Union* is a binary operator that unions the rows of T_{left} and T_{right}. It performs a multi-set union, i.e., it does not remove duplicate rows from the output. The two tables need to have the same number of columns and their types must match. The output table T_2 retains the schema from the left input. We note that different SQL dialects tend to pick different ways of assigning the output schema of a *Union*. We choose one particular style that is closest to SPARKSQL.

A *query* is a sequence of assignments that each produce a new table from existing ones using one the operators described above. Formally, let $T_0, T_1, \cdots, T_n$ be a sequence of input tables. A query is a sequence of assignments of the form $T_i = \texttt{uop}(T_j)$ (for unary operators `uop`) or $T_i = \texttt{bop}(T_j, T_k)$ (for binary operators `bop`) such that $i > n, j < i, k < i$. We sometimes refer to a table T_i by the query that computes it.

3.2 RESIN operators

RESIN introduces two operators, RESINMAP and RESINREDUCE that are used during the optimization process.

RESINMAP $T_2 = \mu[List(\phi, C \leftarrow E)](T_1)$

A RESINMAP is a row-wise unary operator. It is parameterized by a list L of pairs $\langle \phi, C \leftarrow E \rangle$. Its semantics is defined by the imperative code shown in Figure 7. For each input row, the operator can produce up to $\|L\|$ output rows. The operator iterates over L (Line 3), and if the predicate ϕ is satisfied (Line 4) then it applies expressions in E to compute data values of output columns C (Line 5). In other words, RESINMAP applies different chains of *Select* ($\sigma[\phi]$) followed by *Project*$[C \leftarrow E]$ operators, to produce multiple output rows for each input row. This operator requires that for each map $C \leftarrow E$ in its list, the set of output columns C be the same (which is also the schema of the output table). The expressions in E can, however, be different. For example, $\mu[(\phi_1, a \leftarrow e_1, b \leftarrow e_2), (\phi_2, a \leftarrow e_3, b \leftarrow e_4)]$ is a valid operator, whereas the following is not: $\mu[(\phi_1, a \leftarrow e_1), (\phi_2, c \leftarrow e_3)]$.

RESINREDUCE $T_2 = \rho[K, List(\phi, c \leftarrow agg(col))](T_1)$

A RESINREDUCE operator first partitions the input into groups on input columns K, and processes each group independently in a streaming manner. The operator is parameterized by a list L of triples $\langle \phi, c, agg(col)) \rangle$. Figure 8 describes the per-group computation. It takes a single group `G` as input, represented as the partition key `G.key` and a multi-set of rows `G.rows`. It first initializes all the aggregations (Line 3) and then iterates over the rows in `G` (Line 4). For each row, it applies the filter ϕ_i (Line 6) and then updates the corresponding aggregate agg_i (Line 6). Once the entire group is processed, we get a single row containing the keys and the computed aggregates. As notational convenience, we use $init(agg)$ to denote the identity value for an aggregation agg. For instance, $init(sum)$ would be 0, $init(max)$ would be $-\infty$ and $init(min)$ would be ∞.

RESINSIMPLEMAP $T_2 = \lambda[\phi, C \leftarrow E](T_1)$

RESINSIMPLEMAP is a simplified version of RESINMAP that produces at-most one output row per input row. It applies a single predicate ϕ and if a row satisfies the predicate, it computes output columns C by applying expressions E. Its semantics is as in Figure 7 with L having a single element. It represents the most basic form of a mapper that still subsumes a *Select* and a *Project*.

4 RESIN optimizations

RESIN integrates new rules into an existing query optimizer. It leverages the existing rules to perform certain normalizations

that increase the scope of the newly introduced rules. We begin this section by stating rule ordering assumptions and then describe the two core optimizations of RESIN, namely sub-query fusion and binary operator elimination.

4.1 Assumptions

RESIN assumes that the following two rules are applied before further optimizations are attempted. These assumptions are not fundamental to the analysis, they are only required to simplify the presentation.

Column name normalization The query language allows the reuse of column names within a query. For example, $T' = \gamma[a, b \leftarrow sum(b)](T)$ assigns the column name b to the result of an aggregation in T', even though b is already a column in T. We assume that a normalization pre-pass assigns unique names to new columns produced in the query. For example, the above would be rewritten to $T' = \gamma[a, b\#1 \leftarrow sum(b)](T)$. Such a pre-pass is commonly applied by all query optimizers. In addition to aggregations, a *Project* operator can also produce new columns. We require that for any projection map $map(c_i \leftarrow e_i)$, either e_i is just c_i or c_i is a fresh column name. In other words, either a column is just passed through or the output table must use a fresh column name.

Predicate pushdown The optimizer pushes *Select* operators to apply before *Project* operators. Such a rewriting is always possible, and in fact, standard optimizers have many rules that ensure *Select* operators apply on the input data as soon as possible. In particular, RESIN assumes that a *Select* operator is never a parent of a *Project*.

We also define some standard functions. The function $cols(e)$ takes as input an expression e and returns the set of column names used in the expression. For example, $cols(b_1 + b_3 > 0)$ is $\{b_1, b_3\}$. We also define a function $fresh()$ that returns a fresh (globally unique) column name each time. Finally, as the output of a *Union* operator inherits column names from the left argument, we assume the availability of an expression-renaming function $\alpha(\uplus(T_{left}, T_{right}), e)$ that given an expression e over columns of T_{right}, returns an expression over the corresponding columns of T_{left}. For example, if T_{left} has columns (a_1, a_2, a_3) and T_{right} has columns (b_1, b_2, b_3), then $\alpha(\uplus(T_{left}, T_{right}), b_1 + b_3 > 0)$ is $a_1 + a_3 > 0$. We drop the first argument of α when it it clear from the context.

4.2 Generalized sub-query fusion

The goal of sub-query fusion is to combine two queries Q_1 and Q_2 that operate on the same set of input tables, but may produce different outputs. Fusion produces a common query Q and two *residual* RESINSIMPLEMAP operators λ_{r1} and λ_{r2} such that $Q_1 = \lambda_{r1}(Q)$ and $Q_2 = \lambda_{r2}(Q)$. This ensures that the any redundant computation across Q_1 and Q_2 is captured in one single query Q, and only simple map operators (via the residual operators) are needed to get back the original outputs. Furthermore, as would be evident from the way we fuse operators, we ensure that the computation of Q itself does not require more stages than what is required for computing just one of the sub-queries. Finally, we ensure that Q does not output any row that is not needed by either Q_1 or Q_2. This kind of fusion is, of course, not always possible. The rules below define the conditions under which it is possible and how to combine the queries when possible.

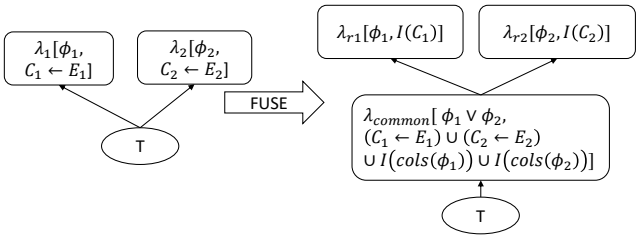


Figure 9: Basic query fusion.

Identity Invariant. Given a RESINSIMPLEMAP operator $\lambda[\phi, map(c_i \leftarrow e_i)]$, we say that it satisfies the *identity invariant* if e_i is simply c_i for all indices i. This means that the operator carries a subset of the input columns unmodified to the output table. For a set of columns C, we use the shorthand $\lambda[\phi, I(C)]$ to represent such operators, where $I(C)$ is the identity function on C: $\{c \leftarrow c \mid c \in C\}$. *We will ensure that all residual operators produced as a result of fusion satisfy the identity invariant.*

4.2.1 Base rule

The rule for fusing two RESINSIMPLEMAP operators applied on the same table is shown in Figure 9. The RESINSIMPLEMAP operators λ_1 and λ_2 apply different filters and projections to the same input table. Fusing these operators is simple, except that we must take care to establish the identity invariant for the residual operators. This is important for recursively fusing more operators up the query tree. The fusion first applies a disjunction of the filters and a union of the projections (λ_{common}). This ensures that the necessary rows and columns are carried forward. Next, all the residual operators λ_1 and λ_2 need to do is to apply the specific filters for Q_1 and Q_2, respectively.

Note that column-name normalization (Section 4.1) guarantees that for any column c, if $c \in C_1$ and $c \in C_2$ then the column must be passed through from T, i.e., both λ_1 and λ_2 apply the projection $c \leftarrow c$. This ensures that the projection map of λ_{common} is well-defined, i.e., it does not include two different mappings for the same output column.

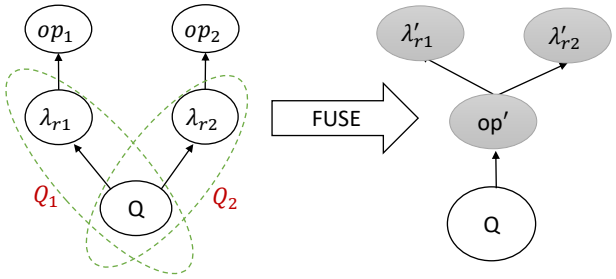


Figure 10: Recursive fusion of unary operators. Given two fusible queries Q_1 and Q_2, shown in dotted circles, the figure shows how to fuse $op_1(Q_1)$ and $op_2(Q_2)$. The shaded circles depict the output of the fusion.

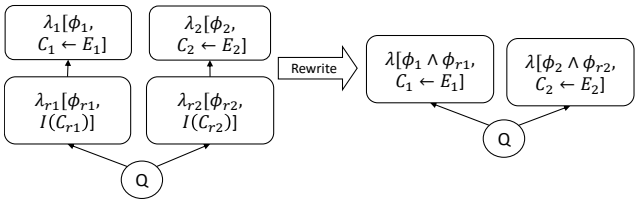


Figure 11: RESINSIMPLEMAP query fusion.

4.2.2 Recursive fusion of unary operators

Fusion proceeds recursively. For this section, fix the fact that $FUSE(Q_1, Q_2) \coloneqq \langle Q, \lambda_{r1}, \lambda_{r2} \rangle$. As described in Figure 10, our goal is to construct $FUSE(op_1(Q_1), op_2(Q_2))$, where op_1 and op_2 are one of RESINSIMPLEMAP (λ), *GroupBy* (γ) or RESINREDUCE (ρ). For ease of notation, an operator λ_x always expands to $\lambda[\phi_x, C_x \leftarrow E_x]$.

Recursive fusion of two RESINSIMPLEMAP operators, which subsumes the fusion of *Select* and *Project* operators, is shown in Figure 11. Observe that predicates ϕ_{r1} and ϕ_{r2} are applied on the output of Q. Further, as the residual operators satisfy the identity invariant, the columns referred in the predicates ϕ_1 and ϕ_2 also come from the result of Q. Therefore, the identity projections in λ_{r1} and λ_{r2} can be dropped and the filters ϕ_1 and ϕ_{r1} can be conjoined together, and so can ϕ_2 and ϕ_{r2}. Fusion then follows by applying the rule in Figure 9.

The rule for fusing two *GroupBy* operators is shown in Figure 12. The figures shows two aggregations on the same table, which is the output of Q, except that they first apply their own filter sλ_{r1} and λ_{r2}, respectively. (For simplicity, we have shown a single aggregation in each of the *GroupBy* operators. The case for multiple aggregations extends easily.) The *GroupBy* operators are on the same key, so we can fuse them into a single RESINREDUCE operator that does the aggregations conditionally as shown in the figure. In addition, the

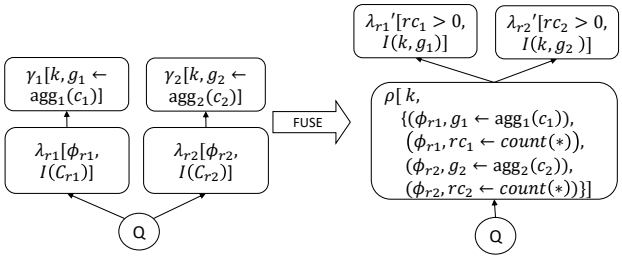


Figure 12: GroupBy query fusion. Both rc_1 and rc_2 are fresh column names.

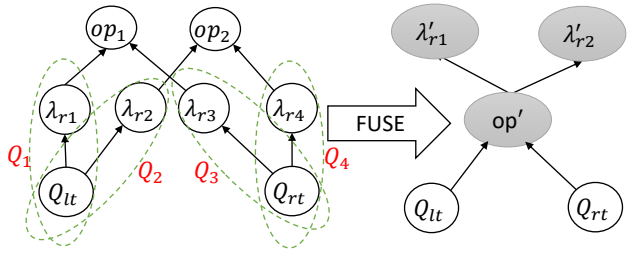


Figure 13: Recursive fusion of binary operators. The figure shows how to extend fusion of two pairs of fusible queries Q_1, Q_3 and Q_2, Q_4 (shown in dotted circles), by additional binary operators op_1 and op_2. The shaded circles depict the output of such recursive fusion.

fusion requires two new aggregations rc_1 and rc_2 that count how often the predicates are satisfied. For the left (respectively, right) group-by to produce any output for a grouping key, at least some rows in the group should satisfy the filter of λ_{r1} (respectively, λ_{r2}). Thus, we need to guard the left (right) output of the fused query with a predicate that ensures that at least one row in the group satisfied the predicate. The new residual operators λ'_{r1} and λ'_{r2} apply the filters $rc_1 > 0$ and $rc_2 > 0$ to only output groups that have at least some rows that satisfy the predicates. The rule extends directly to the fusion of two RESINREDUCE operators as well.

Column Aliasing Our implementation relaxes the rule's precondition that grouping keys be exactly the same; even *aliasing* columns are allowed. That is, columns can be renamed versions of the *same* column in an earlier table. The same relaxation also applies to the join rule that follows later.

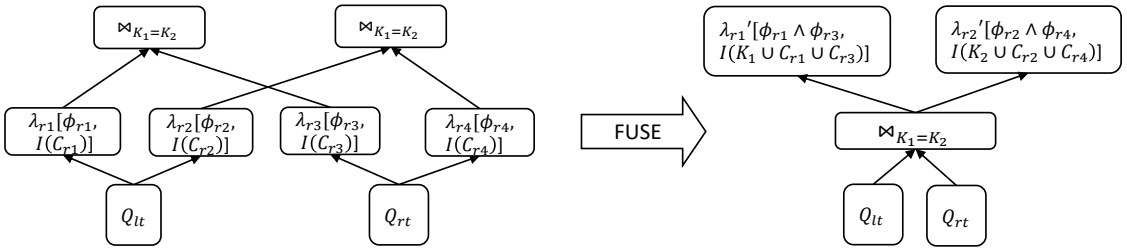


Figure 14: Join query fusion

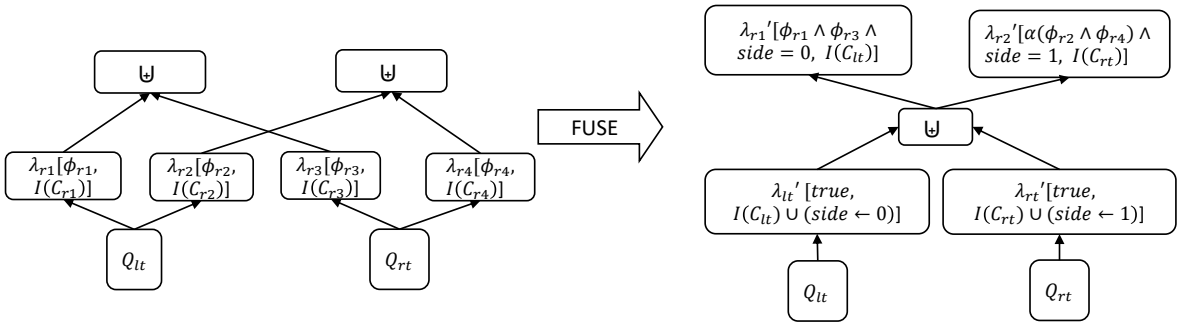


Figure 15: Union query fusion. The column *side* is a fresh name.

4.2.3 Binary operator fusion

Binary operator fusion is depicted in Figure 13. It defines $FUSE(op_1(Q_1,Q_3),op_2(Q_2,Q_4))$ using $FUSE(Q_1,Q_2)$ and $FUSE(Q_3,Q_4)$.

Figure 14 shows the rule for fusing two *Join* operators. The rule simply pulls up the residual predicates from before the join to after. Next, it conjoins the residual predicates that are relevant to $(Q_1 \bowtie Q_3)$, namely ϕ_{r1} and ϕ_{r3}, to obtain λ'_{r1}. Similarly, $\phi'_{r2} = \phi_{r2} \wedge \phi_{r4}$. The residual predicates satisfy the identity invariant. However, we still apply the base fusion rule (Figure 9) to push down the common predicate $(\phi_{r1} \wedge \phi_{r3}) \vee (\phi_{r2} \wedge \phi_{r4})$. This would eliminate rows that are not needed by either $Q_1 \bowtie Q_3$ or $Q_2 \bowtie Q_4$, potentially before a shuffle.

Figure 15 shows the rule for fusing two *Union* operators. We only describe a simplified version of the rule where we assume that Q_{lt} and Q_{rt} are union-compatible, i.e., they have the same number of columns and their types match. This version is enough to cover the core ideas.

In order to fuse two unions, we need to be able to pull up filters above a union. This poses a challenge as the output has rows from both sides and we want to apply different predicates to the rows from each side. To enable this pull up, we add an additional (fresh) column *side* that tags rows with the side that generated them. This additional column is added by applying λ'_{lt} and λ'_{rt} to Q_{lt} and Q_{rt}, respectively. The new residual predicates do an additional check to match rows from the appropriate sides. As the union result renames the columns from the right input, λ'_{r2} additionally applies the renaming function α (defined in Section 4.1).

4.2.4 Operator alignment and exact fusion

The fusion rules described so far only fuse operators of the same type. RESIN also has an auxiliary rule that enables fusion of operators that are preceded by a RESINSIMPLEMAP on one side but not on the other. Given Q_1 and Q_2 are fusible, we enable the fusion of $op_1(\lambda(Q_1))$ and $op_2(Q_2)$, where op_1 and op_2 are fusible according to rules 1-6 above. We do so by adding an empty lambda $\lambda_e = \lambda[true, I(*)]$ as a child of op_2.

We have described the fusion of core SQL operators. Our implementation handles all SPARKSQL operators, but fusion of other operators is only possible if they have the exact same parameters and apply on the exact same query. We define this *exact* fusion rule as $FUSE(op_1(Q_1),op_2(Q_2)) = op_1(Q_1)$ only if $op1 = op2$ and $Q_1 = Q_2$. Finally, note that the rules above define the fusion of two sub-queries. Through repeated application of the rules we can fuse any number of sub-queries (say, n) into a single query with n residual operators.

4.3 Binary operator elimination

When the two arguments of a binary operators can be fused, RESIN can sometimes eliminate the binary operator altogether. The two elimination rules are defined below.

UNION ELIMINATION RULE

Given a *Union* query $\uplus(Q_1,Q_2)$ where Q_1 and Q_2 can be fused such that $FUSE(Q_1,Q_2) := \langle Q,\lambda_{r1},\lambda_{r2}\rangle$ then we

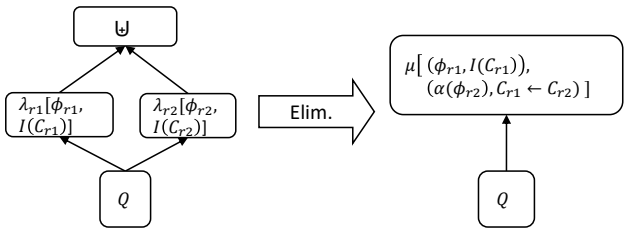


Figure 16: Union elimination rule.

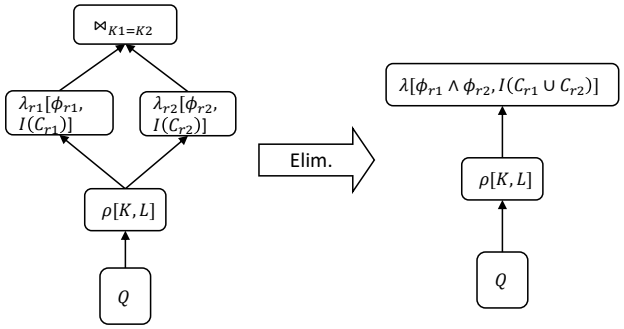


Figure 17: Join elimination rule. The rule requires that each of K_1 and K_2 alias with K.

can eliminate the *Union* altogether using the rule shown in Figure 16. The rule follows directly from the definition of ResinMap. Recall that a ResinMap operator $\mu[List(\phi, C \leftarrow E)]$ can produce multiple outputs per input row. This sufficient to implement the *Union* operator of the form above. The resulting ResinMap operator has one entry for each input that applies the filter ϕ_{ri}. The right expressions are renamed using α and assigned to the column names from the left (C_1) to conform to the semantics of a *Union*. Figure 2 is an example application of this rule.

Resin rules show that any single-input query consisting of *Select*, *Project* and *Union* operators can be implemented by a single ResinMap operator.

Join elimination rule The goal of this rule is to substitute a binary join operator with a mapper, which is a row-wise unary operator. This is only possible if the output of the join has already been computed in the fused query. This holds when the join combines the results of a ResinReduce query $\rho[K, L]$ and is equi-join on K (modulo aliasing). The rule is shown in Figure 17. Figure 5 shows an example application of this rule.

5 Implementation

We integrated Resin into a popular state-of-the-art big-data system Spark [26]. Our optimizations are general and can be applied to other big-data systems [22, 30] as well. We chose Spark because it is easier to extend [5], has rich code-gen support as well as competitive performance. Moreover, Spark already performs some low-level I/O optimizations. For instance, it implements exchange reuse [1, 2] that determines if two exchanges are exactly equivalent and skips the duplicate computation. It also implements store-predicate pushdown that pushes down filters and projections to the storage layer [3].

Spark makes use of the Catalyst query optimizer [5]. Optimization rules in Catalyst are organized into batches. As is standard, logical rules are applied before physical rules. Each physical operator has a pre-defined map-reduce implementation based on a low-level *resilient distributed dataset* (RDD) API [25]. Spark uses a whole-stage code generator [23] to efficiently compile all operators in a single stage. We describe key details of our implementation.

Initiation and termination of Resin rules We added all Resin rules in a batch that executes after the standard optimizations are applied. These rules apply in a single (pre-order) traversal of the query tree. Resin initiates fusion starting from input table scans. It then moves up the tree fusing operators recursively. The fusion process terminates when none of the fusion rules apply. At this point, Resin applies the operator elimination rules in cases where the consumers of a fused query share a common parent. After elimination, the resulting query could have zero or more fused sub-queries whose output is consumed more than once, requiring the use of *exchange* operators, as described next.

Resin exchange reuse The only operator in Spark whose output can be consumed more than once is an *exchange* operator. Thus, Resin introduces an exchange at the reuse points. An exchange is parameterized by a partitioning column. To decide on the partition column, Resin traverses up along each of the consumers C_i until it hits an operator that requires partitioning (ResinReduce, *Join*, *GroupBy*), and identifies a partitioning column p_i for each consumer. Next, it picks the column p_i that is required by most consumers (we use random choice to break ties).

Resin operators We added three new logical operators with the structure defined in Section 3. We also add their corresponding physical operators. The physical operator for ResinSimpleMap is just a combination of *Select* and *Project*. We added a new physical operator that implements ResinMap with appropriate whole-stage code-generation support. The physical operator for ResinReduce is implemented by carefully extending existing aggregation iterators in Spark. This allowed us to delegate the handling of different column types and the various associated subtleties in the application of aggregation functions (e.g., *null* values, type-casting, overflow/underflow, etc.) to routines already present in Spark. Finally, we added implementation strategies for our opera-

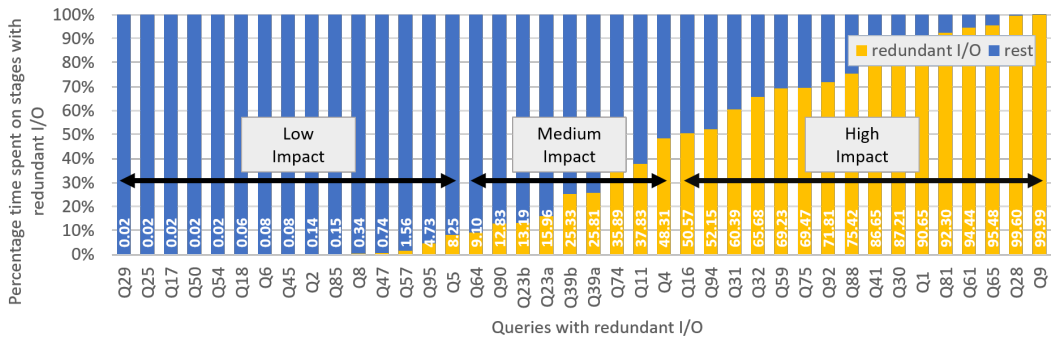


Figure 18: Fraction of time spent in stages with redundant I/O.

tors. The strategies analyze the logical operators, construct partial aggregates and introduce partitioning operators (for RESINREDUCE), and substitute the logical operators with corresponding physical operators.

6 Evaluation

We evaluated RESIN using the TPCDS benchmark suite, consisting of 104 queries, at scale factors of 1TB and 10TB. The evaluation was done on two different SPARK clusters. We used a cluster with 120 cores and roughly 480GB memory, spread over 10 nodes for evaluating at 1TB scale. For evaluating at 10TB we used a cluster with 480 cores and 1.6TB memory, spread over 34 nodes. The input tables were stored in parquet format. We ran each query 5 times, discarded the first run and took average of the rest. Among the 104 queries, we found that 40 queries have redundant I/O. As mentioned before, the baseline already has basic I/O optimizations. It pushes predicates and projects to the store for all these queries. And it is able to reuse exchanges (usually right after a map stage) even without RESIN optimizations in about half of these queries. In the rest of this section, we focus on these queries alone. We begin by presenting detailed results at 1TB scale and present summary results at 10TB scale in Section 6.4.

6.1 Optimization opportunity

For each query, we identified stages that perform redundant I/O. This was done post-facto by comparing baseline and optimized plans, and determining the baseline stages that were fused together by RESIN. Figure 18 shows the fraction of time spent in these stages relative to the total execution time of the query. The larger the fraction, the greater the optimization opportunity. We find that 40% of the queries spend at least 50% of the time in stages with redundant I/O. We mark these queries as *high-impact* queries as they have significant potential for improvement. Another 25% spend at least 10% of their time in stages with redundant I/O, and we mark them as *medium-impact* queries. The remaining (*low-impact*) queries may have some redundant I/O but eliminating it is unlikely to affect the overall query execution time.

TPCDS queries are over multiple (*fact* and *dimension*) input tables. There are 6 large fact tables and several small dimension tables. A deeper inspection of our results revealed that the fraction of time spent in redundant sub-queries is significantly influenced by whether one of these large tables was redundantly processed or not. All queries that have medium or high impact were processing at least one such table multiple times (sometimes even after joining with few other tables).

6.2 Speedup from RESIN optimizations

Figure 19 reports the performance improvements from RESIN on high and medium impact queries. These cover 25% of the entire benchmark suite. As can be seen, RESIN improves the execution time of most of the queries. It achieves an average (geomean) speedup of 1.4× across these queries. RESIN performs particularly well on *high-impact* queries where it achieves a geomean speedup of 1.6× with some queries speeding up by 6×.

The queries that benefit most ($Q9$, $Q28$, $Q88$, $Q75$, $Q31$, $Q90$) are also ones where RESIN was able to apply *binary operator elimination*. All the other queries benefit only from *generalized sub-query fusion*. Some of these ($Q65, Q61, Q81, Q1, Q30, Q59$) had multiple exchanges after fusion on the *reuse exchange column* and they see moderate gain. A few queries ($Q92$, $Q32$, $Q16$,$Q41$) had reuses close to input scans. These are the queries that see the least benefit because the baseline already performs some basic I/O optimizations (exchange reuse and store-predicate pushdown; see Section 5).

In two queries ($Q74, Q41$) the data overlap between the sub-queries that were fused was very low. However, fusion still helps produce execution plans with fewer stages, and does so while guaranteeing that the number of rows shuffled after fusion is no more than the baseline. We find that, in $Q74$, simplifying the plan has some second order system effects (see Section 6.3), and fusion improves performance. In $Q41$,

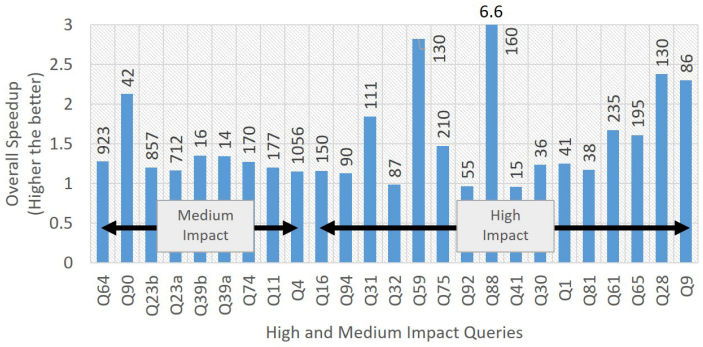


Figure 19: Overall execution time speedup for high and medium impact queries. Each bar is labeled with the execution time of the baseline query (in seconds).

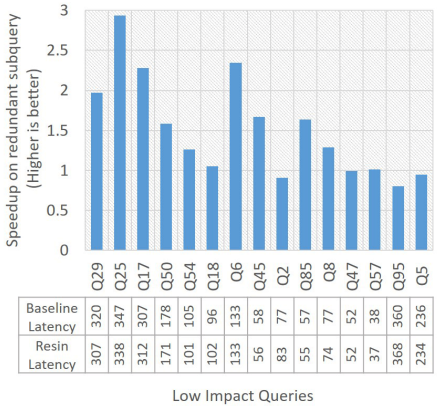


Figure 20: For low impact queries we plot the speedup in sub-query with redundant I/O alone. Along the x-axis we report the execution time of the entire query (in seconds) with and without RESIN.

the reuse is close to the input and hence fusion only eliminates one map stage. As a result, we see a small 3% degradation.

Comparison with BLITZ We evaluated BLITZ on these queries and found that it only optimizes two of the queries: *Q*9 and *Q*28. Both these queries perform a chain of joins at the end. BLITZ was only able to eliminate the first of these joins and therefore was only able to get speedups of 1.6× and 1.9×, respectively. This limitation has also been acknowledged in prior work [10]. RESIN eliminates multiple joins and achieves a speedup of 2.4× and 3.3×, respectively, on these queries.

Speedup on low impact queries Figure 20 reports speedups for low impact queries. We report the execution time of the entire query along the x-axis. As can be seen RESIN optimizations have no significant gains or degradation on any of these queries. To isolate the effects of RESIN optimizations, we plot the speedup for the sub-query that was optimized. RESIN achieves a moderate speedup on several of these sub-queries. RESIN optimizations show a small degradation in a few of these sub-queries (*Q*2, *Q*5, *Q*95). In *Q*5 the amount of redundant I/O is too small to matter. In *Q*2, *Q*95, the baseline already performs an exchange reuse. RESIN fuses one additional operator, but once again the additional I/O is too small to matter.

Overall, RESIN reduces the total time to run all the 104 queries by 12%. Note that RESIN has a negligible impact on query optimization time; the overall compilation time for the entire benchmark increased from 42 to 45 seconds.

6.3 Impact of RESIN optimizations on systems resources

Figure 21 - Figure 24 plot the impact of RESIN optimizations on disk, network, memory and CPU for medium and high impact queries (we see no discernible impact on low impact queries). For disk, we report the cumulative bytes of data accessed from disk. For network, we report the cumulative number of packet transfers performed. Note that data sizes transferred over the network follow the same trend as disk I/O, as most I/O in a big-data setting is over the network. For memory, we plot the cumulative memory footprint. For CPU, we plot the total CPU time spent by all tasks on all machines. This is a measure of the total CPU work done to evaluate a set of queries and is largely independent of cluster size [19]. We infer the following conclusions from these plots.

First RESIN reduces the cumulative CPU, network and disk footprint, consuming 24%, 25% and 19% fewer resources respectively. The savings in-terms of CPU are slightly higher than disk because RESIN not only saves on I/O but also on I/O induced processing (compression, serialization etc) which have a significant compute cost [14].

Second, RESIN achieves these benefits while incurring the same overall memory cost (Figure 23) as the baseline. A few queries (*Q*64, *Q*31, *Q*61) see a slight increase in memory requirement, while a few others (*Q*4, *Q*75, *Q*88) need lesser memory. However, all these queries see significant reduction in execution time. Overall even if fusion increases the amount of data processed by each operator, it does not impact the overall memory footprint of the workload (see Figure 19).

Third, the gap between RESIN and the baseline widens as we move to the right. Queries on the right usually have deeper operator trees and this graph demonstrates that RESIN is able to fuse deep and complex queries.

Finally, the plots indicate that RESIN optimizations

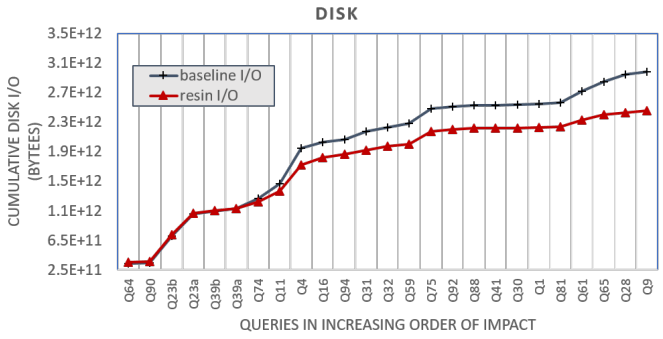


Figure 21: Cumulative disk I/O

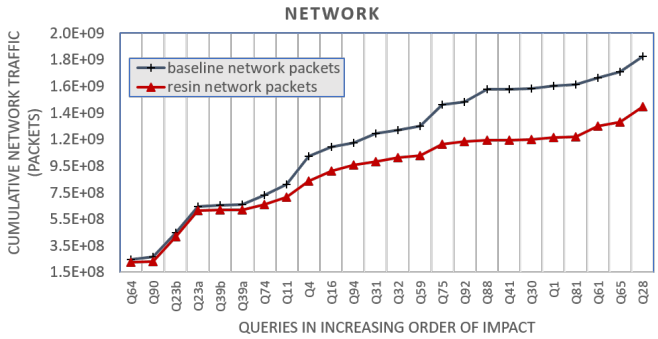


Figure 22: Cumulative network packets

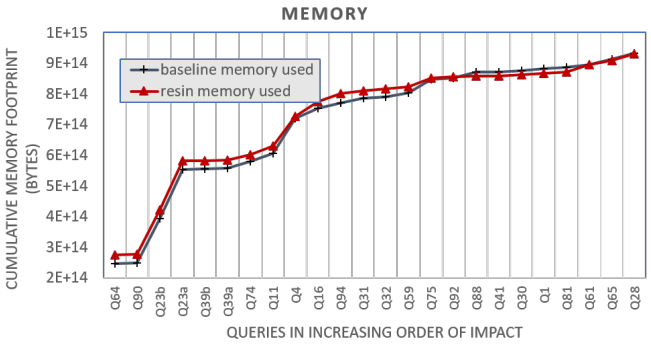


Figure 23: Cumulative memory footprint

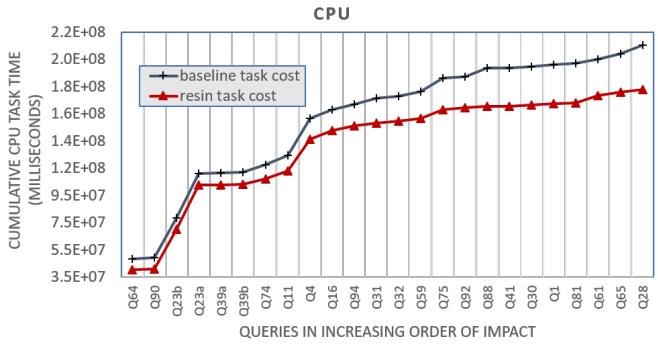


Figure 24: Cumulative CPU time of tasks

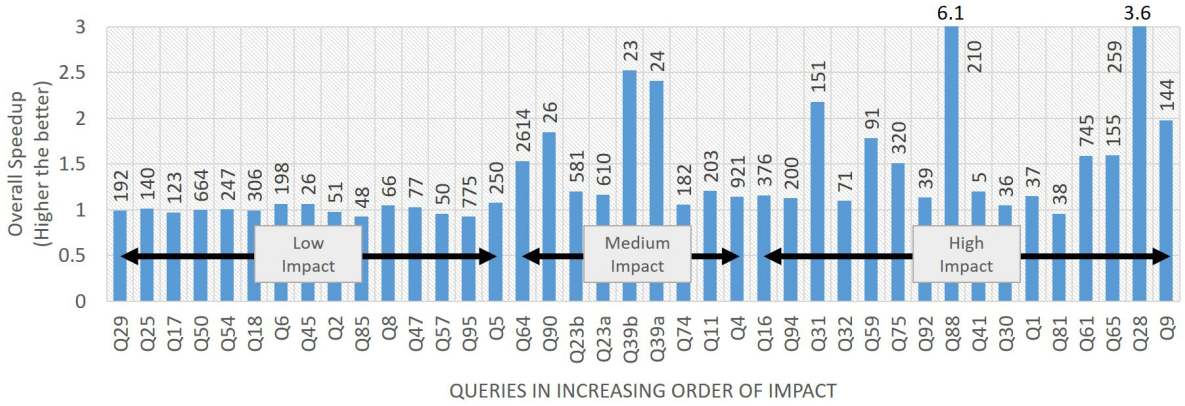


Figure 25: Speedup for 10TB TPCDS. Each bar is labeled with the execution time of the baseline query (in seconds).

are fairly robust, even the worst performing queries ($Q92$,$Q32$,$Q41$) do not show any discernible degradation on any of the system metrics. In $Q74$ RESIN fusion does not reduce the amount of disk I/O, but it still reduces the CPU and network load, and hence sees an execution time benefit.

6.4 Impact on larger scale data

We report the impact of RESIN on TPCDS at 10TB scale. Figure 25 shows the speedup's obtained for the 40 affected queries. We see that RESIN does somewhat better at larger scale. It obtains higher speedup on a few medium and high impact queries ($Q64, Q39a, Q39b, Q28$) while achieving similar speedups for the other queries (except $Q59$). We find that the average (geomean) speedup for high and medium impact queries goes up to 1.5× (was 1.4× at 1TB). Once again, the optimizations have no significant improvement or degradation on the low-impact queries. Figure 26 reports the I/O savings. The total disk I/O saved went up to 31% (was 19% at 1TB). Overall, RESIN reduces the execution time of the entire workload (104 queries) by 17%.

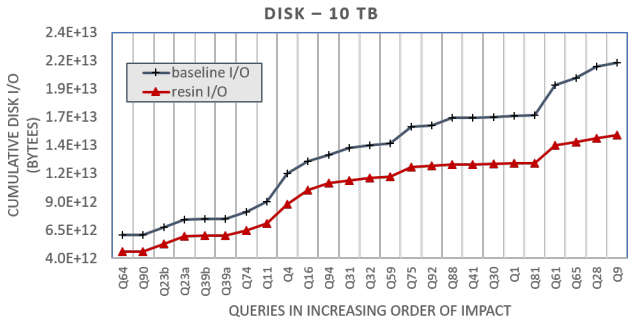


Figure 26: Cumulative disk I/O for 10TB TPCDS.

7 Related Work

We discuss three broad lines of work related to this paper.

Advances in big-data query optimization Big-data query optimizers borrow and build upon rewrite rules from the database literature. Several big-data-specific optimizations have also been used [8, 9, 15, 16, 28–30]. However, none of these logically fuse multiple operators or eliminate binary operators. The work that is most closely related to RESIN is BLITZ [10], which presented an extension to the query optimizer to find and substitute sub-queries that can be implemented by a streaming operator. BLITZ added new rules that optimize three specific query patterns. Two of these patterns were self-joins and self-unions that followed a *GroupBy* on the same input table. The third pattern was a specialized implementation of a *min* aggregation followed by a *Join*. The BLITZ rules can perform some of the operator eliminations that RESIN can perform. However, we find that BLITZ patterns cover a very small fraction of queries in standard benchmarks. Only one of the patterns applies to TPCDS queries and that too only on two queries. Furthermore, BLITZ operators do not compose with each other and therefore do not even eliminate redundant shuffles from multi-way self-joins and self-unions. RESIN introduces the ability to fuse multi-input sub-queries and eliminate unnecessary shuffles. This fusion facilitates more join and union elimination.

Multi-query optimization Multi-query optimization (MQO) is a well studied problem in classical database literature [11, 17, 20, 31]. The goal of MQO is to optimize many concurrently submitted queries together, and is typically done by reusing results of common sub-queries. Such optimizations are typically performed in a single scale-up database setting and trade-off latency for throughput. The goal of RESIN is very different. RESIN looks for intra-query redundancy in the big-data setting, and eliminates it while ensuring no additional rows are shuffled. Thus, it simultaneously improves both latency and throughput.

The fusion techniques proposed here are also significantly different than MQO. MQO is typically limited to Select-Project-Join (SPJ) queries, whereas RESIN supports composible fusion for all SparkSQL operators. Such support is necessary to eliminate redundancy from deep queries. Our evaluation reveals that optimization of the high and medium impact queries in TPCDS requires fusion of a large number of operators: 21 of 25 queries have 10 to 30 operators. We show that fusion and elimination are not always possible without having new operators and propose RESINMAP and RESINREDUCE operators to enable this. For example, *Union* elimination is only possible with RESINMAP and *GroupBy* fusion is only possible with RESINREDUCE. Finally, our binary operator elimination rules are not part of any multi-query or database optimizer.

Code generation techniques for query processing. There is a long line of work on compilation techniques to generate efficient single-machine code for a chain of SQL operators [6, 12, 13, 23]. Such compilers target low level inefficiencies such as virtual call overheads and computation of common sub-expressions across operators. This is an active area of research, and includes recent efforts like FLARE [6] that target the compilation of SPARK to single machine systems. Such compilers have limited scope in the big-data setting because they only optimize the code within a single stage [6]; determining what operators constitute a stage is still decided by the query optimizer. SPARK makes use of one such code-generation engine [23] that builds upon HyPer [13]. The physical operators that we add are whole-stage code-gen enabled and benefit directly from such techniques.

Recent literature has seen advance techniques that optimize mixed-mode queries: queries that embed non-SQL functions and expressions into SQL [7, 16, 24]. This line of work is orthogonal to RESIN.

8 Conclusions

The cost of running big-data queries is dominated by I/O. This paper proposes RESIN, a system that helps identify and eliminate redundant I/O. The system proposes extensions to big-data query optimizers that enable first class map-reduce reasoning during query compilation. We show how these can be used to fuse operators processing overlapping data into a single stage of computation, and sometimes eliminate expensive binary operators altogether. We demonstrate that the optimizations are useful for 40% of queries in TPCDS, and bring significant gains (average 1.4×) to a quarter of the benchmark queries.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Wenguang Chen for their valuable feedback and suggestions. We would also like to thank Ajith Shetty, Srinivas T, Shahid K, Lev Novik and Tomas Talius for code and design reviews.

References

[1] Reuse Query Fragments. https://issues.apache.org/jira/browse/SPARK-13756, 2016.

[2] Reuse the exchanges in a query. https://issues.apache.org/jira/browse/SPARK-13523, 2016.

[3] Parquet Predicate Pushdown improvement. https://issues.apache.org/jira/browse/SPARK-25419, 2018.

[4] Amazon red-shift. https://docs.aws.amazon.com/redshift/index.html, 2020.

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[6] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In OSDI, pages 799–815, 2018.

[7] X. Fan, Z. Guo, H. Jin, X. Liao, J. Zhang, H. Zhou, S. McDirmid, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. IEEE Transactions on Parallel and Distributed Systems, 26(6):1718–1731, June 2015.

[8] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In OSDI, pages 121–133, 2012.

[9] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major technical advancements in apache hive. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, pages 1235–1246, New York, NY, USA, 2014. ACM.

[10] Jyoti Leeka and Kaushik Rajan. Incorporating super-operators in big-data query optimizers. PVLDB, 13(3):348–361, 2019.

[11] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Mqjoin: Efficient shared execution of main-memory joins. Proc. VLDB Endow., 9(6):480–491, January 2016.

[12] Derek Gordon Murray, Michael Isard, and Yuan Yu. Steno: Automatic optimization of declarative queries. In PLDI, pages 121–131, 2011.

[13] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. PVLDB, 4(9), 2011.

[14] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pages 293–307, Oakland, CA, May 2015. USENIX Association.

[15] Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, and Jaliya Ekanayake. Hyper dimension shuffle: Efficient data repartition at petabyte scale in. PVLDB, 12(10):1113–1125, 2019.

[16] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In SOSP, pages 153–167, 2015.

[17] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, page 249–260, New York, NY, USA, 2000. Association for Computing Machinery.

[18] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David G Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Allen Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed Aly, Divy Agrawal, Ashish Gupta, and Shivakumar Venkataraman. F1 query: Declarative querying at scale. pages 1835–1848, 2018.

[19] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. Optimizing big-data queries using program synthesis. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 631–646, New York, NY, USA, 2017. ACM.

[20] Timos K. Sellis. Multiple-query optimization. ACM Trans. Database Syst., 13(1):23–52, March 1988.

[21] Srinath Shankar, Rimma Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David DeWitt, and César Galindo-Legaria. Query optimization in microsoft sql server pdw. In Proceedings of the 2012 ACM SIGMOD International Conference on

Management of Data, SIGMOD '12, page 767–776, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. Proc. VLDB Endow., 2(2):1626–1629, August 2009.

[23] Reynold Xin and Josh Rosen. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. https://tinyurl.com/mzw7hew, 2015.

[24] Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, Huan He, and Haibo Lin. Niijima: Sound and automated computation consolidation for efficient multilingual data-parallel pipelines. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19, pages 306–321, New York, NY, USA, 2019. ACM.

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[27] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, pages 43:1–43:15, New York, NY, USA, 2018. ACM.

[28] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, pages 43:1–43:15, New York, NY, USA, 2018. ACM.

[29] J. Zhou, P. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In ICDE, pages 1060–1071, 2010.

[30] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. Scope: Parallel databases meet mapreduce. The VLDB Journal, 21(5):611–636, October 2012.

[31] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, page 533–544, New York, NY, USA, 2007. Association for Computing Machinery.



A Simpler and Faster NIC Driver Model for Network Functions

Solal Pirelli and George Candea, EPFL

Abstract

The advent of software network functions calls for stronger correctness guarantees and higher performance at every level of the stack. Current network stacks trade simplicity for performance and flexibility, especially in their driver model. We show that performance and simplicity can co-exist, at the cost of some flexibility, with a new NIC driver model tailored to network functions. The key idea behind our model is that the driver can efficiently reuse packet buffers because buffers follow a single logical path.

We implement a driver for the Intel 82599 network card in 550 lines of code. By merely replacing the state-of-the-art driver with our driver, formal verification of the entire software stack completes in 7x less time, while the verified functions' throughput improves by 160%. Our driver also beats, on realistic workloads, the throughput of drivers that cannot yet be formally verified, thanks to its low variability and resource use.

Our code is available at *github.com/dslab-epfl/tinynf*.

1. Introduction

The networking world is moving from hardware network functions to software ones to gain flexibility. This brings new problems to light in the network stacks of mainstream operating systems, which were not designed for this use case. In response to this move, the kernel-bypass model for software networking appeared, designed for low latency and high throughput. However, one area of the stack that remains under-explored is network drivers. We present the state of network functions, stacks and drivers in Section 2.

Modern network cards contain powerful and complex hardware offloads, but their core features are conceptually simple. Network cards fetch requests and return responses to software using data structures named descriptors. The main complexity for packet reception and transmission is the descriptor ownership mechanism. We present the basics of modern network cards in Section 3.

The current network driver model is too flexible for the needs of common network functions, which must pay the complexity costs of modern drivers without reaping their benefits. This is mainly because the current driver model allows network functions to process packets out of order, a powerful feature that is not needed in many of the core functions making up the Internet's backbone. We formalize the current driver model for network cards and propose our conceptually simplified version in Section 4.

We implement our new driver model for the Intel 82599, a modern 10 Gb/s Ethernet controller. Our implementation uses the model's insights and stays as simple as possible: it is only 550 lines of C code. Its key features are a minimal number of operations thanks to the driver design and to modern network card features, as well as some simple but powerful scheduling algorithms. We present our driver, which we call "TinyNF", in Section 5.

This paper's core hypotheses are that our simpler model (1) makes network functions easier to formally verify, (2) is faster than the current most complex driver model that can be formally verified, (3) provides competitive performance against the fastest state-of-the-art drivers regardless of complexity, and (4) is applicable to most network functions that are deployed today.

We show that hypotheses (1) and (2) hold in Section 6. Our driver has exponentially fewer code paths than current drivers and can thus be used to formally verify network functions in 7x less time than with a state-of-the-art driver while offering 2.5x the throughput, as well as lower median and tail latency.

We show that hypothesis (3) holds in Section 7, with the surprising observation that our driver outperforms the state of the art using real network functions even though it loses on a synthetic "no-op" function. This is because our driver slows down less when running real functions due to having room to grow in instruction-level parallelism and cache use.

We provide evidence for hypothesis (4) in Section 8, showing that our model is applicable to most of the low-level network infrastructure, either running on bare metal or as virtualized network functions.

We believe that the separation in common use between "drivers" and other software is blurry, and we argue that it hinders progress. This situation is worse in networking due to the lack of good baselines for benchmarks, leading to driver optimizations that increase complexity but may not increase performance in the real world. Our minimal driver also highlights opportunities in hardware documentation. We discuss these issues in Section 9.

In summary, we make the following contributions: (1) a simplified driver model for network functions that process packets in order, (2) a formally verified driver based on our model that is easier to reason about and faster on realistic workloads than existing drivers, and (3) evidence that the current standard of benchmarking for network drivers leads to suboptimal performance in the real world.

2. Background on network functions

In this section, we introduce network functions: packet-processing appliances performing tasks such as routing, rate limiting, access control or caching.

Hardware network functions are the traditional way to implement network functions for high-traffic networks. They are physical boxes with custom hardware that are part of the network, distinct from standard computers.

They are typically robust because fixing hardware bugs after deployment is not possible, thus engineers must test them extensively before deployment. However, they are not flexible because they cannot be modified after deployment. Changing a network's policies can require replacing the hardware entirely.

Software network functions run on general-purpose hardware such as x86 and mainstream operating systems such as Linux, communicating with network cards through a software stack that includes drivers and implementations of protocols such as IP and TCP.

The networking world is moving to software network functions to increase flexibility. Software network functions are flexible since they have low deployment costs. This means correctness guarantees, while important, are not a hard requirement for deployment.

Verifying the correctness of software network functions is an open problem, with recent work showing it is easier than the intractable problem of general software verification. The Vigor [33] project verifies network functions without human interaction, but cannot deal with common optimizations such as parallelism or batch processing.

The other key concern of software network function is performance. This includes variability, since worst-case performance determines the guarantees network operators can offer. These guarantees turn into business concerns such as Service Level Agreements.

To illustrate how crucial performance is, consider the time budget for processing a 64-byte packet and its 20-byte Ethernet header at 10 Gb/s: $(64+20) * 8 / (10*10^9) = 67.2$ns. This is the same order of magnitude as a memory read; a network function will exceed its time budget if it needs data outside the CPU cache.

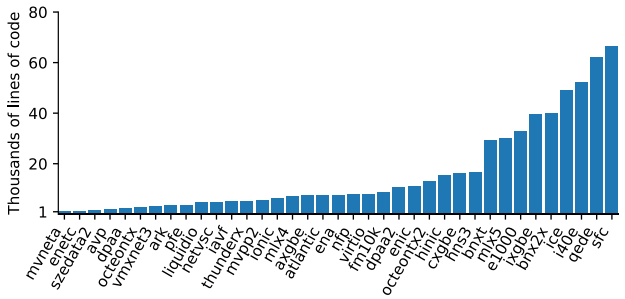


Figure 1. Number of lines of C code in the network drivers included with DPDK 20.02.

Software network stacks in modern operating systems are not adapted to network functions for three reasons.

First, traditional stacks use a push model: hardware uses interrupts to notify software of packet reception. If packets are infrequent, this is efficient. But in network functions, packets are frequent thus interrupt overheads dominate. The pull model, in which software continuously polls for packets, better fits network functions because it is efficient if most polls succeed, as is the case under high load.

Second, traditional stacks only access hardware through the operating system to provide isolation. Going through the operating system is an expensive operation, especially given the low time budget for each packet. But network functions typically run alone, paying the performance cost of isolation without the associated benefits. Systems such as netmap [28] have shown this cost can be amortized by processing packets in batches.

Third, traditional stacks allow to manage packet buffers with complete flexibility. This is convenient for general-purpose programs but hinders optimizations in the network stack. Network functions have restricted and well-defined behavior, yet they pay the performance cost of flexibility. Systems such as Windows Registered I/O [24] have shown that decreasing flexibility can increase performance.

Kernel-bypass stacks, which allow programs to access hardware directly instead of going through the operating system, arose from the need for different tradeoffs. These stacks also focus on polling instead of interrupts, on tighter control of packet buffers, and on processing packets in batches. The de facto standard kernel-bypass stack is DPDK, the Data Plane Development Kit [5].

Drivers, network or otherwise, have a poor reputation among software developers because of the challenges of hardware interactions and the lack of documentation.

Developers can only rely on specifications released by manufacturers to know how hardware behaves, and these specifications are not always public. Reverse-engineering hardware is infeasible without special equipment, unlike software. Since drivers are often exclusively maintained by hardware manufacturers, driver developers do not need to publicly document their code. Bug-finding efforts have shown that driver code is far from bug-free [21, 25].

These problems lead developers to think of drivers as mystical black boxes. But drivers are a fundamental part of the network stack; their correctness and performance are upper bounds on the entire stack.

As an example of driver complexity, the network drivers in DPDK, which supports many different types of hardware, all have at least 1,000 lines of code, as we show in Figure 1, with the largest one being over 66,000 lines of code.

Emmerich et al. [10] showed that network drivers can fit in under 1,000 lines of code, though their driver focuses on educational value and not performance or correctness.

3. Background on network cards

In this section, we summarize the architecture of modern Network Interface Controllers, or "NICs" for short, which is necessary to understand network driver design.

While NICs are diverse, the core concepts are similar. We estimate that, out of the 44 families of physical or virtual NICs supported by DPDK 20.02, this section applies to 40 of them. The remaining ones are three FPGA-based cards and one proprietary virtual NIC.

Communication between CPU and NIC uses three channels: PCI registers, NIC registers, and RAM.

PCI registers are stored on the NIC and accessed by the CPU using port-mapped I/O. The CPU only uses them for the first stage of NIC initialization.

NIC registers are stored on the NIC and accessed by the CPU using memory-mapped I/O. Their latency is an order of magnitude higher than RAM [19], making them a performance bottleneck.

RAM is the main shared storage. The CPU accesses it as usual, and the NIC uses Direct Memory Access, or "DMA" for short, to transfer data into it. RAM holds packet buffers and metadata. The CPU and the NIC are not notified when the other has modified RAM; if they want to be aware of changes, they must poll RAM or use a side channel.

The packet descriptor is the main NIC data structure, containing a pointer to a data buffer and some metadata. The metadata typically contains required fields such as the packet length, and optional fields such as whether to use advanced hardware offloading features.

Software chooses the total number of descriptors when initializing hardware. Descriptors are given from the CPU to the NIC to issue commands, such as packet transmission, and given by the NIC back to the CPU when the associated command has finished. Different NICs have different ways to manage descriptor ownership, such as flags in metadata.

Software can change the buffer pointer before giving a descriptor to the NIC. This lets developers implement buffer pools, to reuse descriptors without losing received data. This is useful for cases such as TCP, where packets must be kept until an entire message has arrived.

Reception and transmission are the core operations of network cards, and work in symmetric ways.

To receive packets, the CPU gives descriptors to the NIC indicating where to deposit packets in memory. The NIC gives descriptors back when it has received packets. The NIC sets descriptor metadata to indicate the packet length and other such information.

To transmit packets, the CPU gives descriptors to the NIC indicating where packets are in memory, and the NIC gives descriptors back once it has transmitted packets. The metadata is set by the CPU, to inform the NIC of the packet length and other such information.

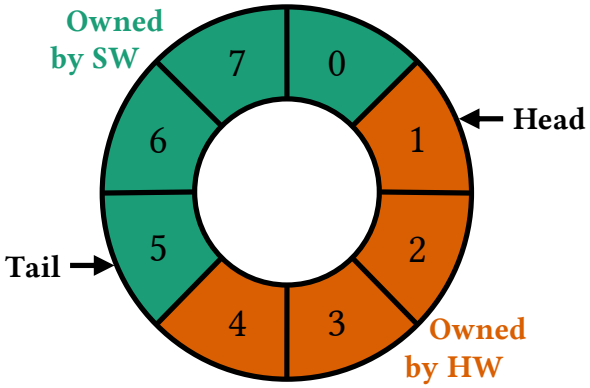


Figure 2. A descriptor ring with 8 elements; the head is 1 and the tail is 5, thus hardware owns elements 1, 2, 3 and 4.

Descriptor rings are the main mechanism for descriptor ownership in modern cards. We present here their inner workings in Intel's 82599 NIC as a concrete example.

A descriptor ring is composed of a region of memory, a head pointer, and a tail pointer. The memory is in RAM, while the pointers are NIC registers. Descriptors between the head, inclusive, and the tail, exclusive, belong to the NIC. Other descriptors belong to the CPU. If the head and tail are equal, the CPU owns all descriptors. We present an example ring in Figure 2.

Since descriptors start in an unknown state, descriptor metadata has a "Descriptor Done" flag to let the CPU know whether a descriptor has been processed by the NIC or was just never initialized.

The CPU gives descriptors to the NIC by clearing their "Descriptor Done" flag and incrementing the tail pointer. Since the tail pointer is a NIC register, the NIC immediately notices the change. The NIC gives descriptors back to the CPU by setting the "Descriptor Done" flag in metadata and incrementing the head pointer. To know when a descriptor has been given back, the CPU polls the metadata.

The head and tail can only be incremented, though they can be incremented by more than 1 to give descriptors in batches. Decrementing is forbidden since it would logically be an attempt to steal descriptors.

NIC queues are a hardware mechanism to allow for parallel packet processing. A queue consists of a descriptor ring and some configuration. The NIC places all received packets in the first reception queue by default; developers can configure the NIC to route packets to a queue based on packet headers, such that packets belonging to the same logical flow are routed to the same queue.

For transmission, all queues behave in the same way, without flow tracking: packets added to any transmission queue are sent to the wire regardless of which queue it is.

Queues allow multiple CPU cores to handle packets without having to synchronize NIC accesses, increasing software scalability.

4. Simplifying the driver model

In this section, we present the existing kernel-bypass driver model, and our proposed simplification of it for common network functions.

The driver model in modern frameworks such as DPDK is based around reusing a fixed set of packet buffers, in order to avoid the overheads of memory allocation. We formalize this model as a diagram in Figure 3, where the overall system is a set of first-in-first-out buffer queues. Each conceptual "step" of the system is performed by one of three actors: the NIC, the driver, or the network function. In the initial state, all buffers are in the "Free" state, which represents unused buffers in the buffer pool.

The driver typically takes the first steps, by "allocating" buffers from the pool and giving them to the NIC for reception. This "allocation" refers to taking buffers from the pool, not creating new ones. If there are buffers in the "receiving" state, the NIC can transition them to the "received" state once it gets data from the network. The network function typically runs a polling loop to move buffers into the "processing" step. From there, the network function can choose to transmit the buffer, possibly after modifying it. The NIC will then send out the buffer contents to the network and move the buffer to the "transmitted" state. The driver moves transmitted buffers back to the "free" state at well-defined points, for instance when there are too few free buffers left. The network function can also choose to keep the buffer for later, or to "drop" it and return it to the pool. The network function can also allocate buffers from the pool and process them like received buffers.

Unlike classical driver models found in mainstream operating systems and exposed to programmers in libraries such as BSD sockets [29], the system is closed: none of the actors can insert buffers into the system from the outside, such as by asking the operating system for memory. Actors cannot remove buffers either, though the network function is allowed to keep buffers indefinitely by using its "keep" transition to reorder buffers in the "processing" state.

The reason for a closed system is performance: buffer allocation and deallocation are expensive. This is not only due to general software issues such as the overheads of keeping a "free list" of memory blocks, or the cost of asking the operating system for more memory, but also to an issue specific to drivers: memory pinning. The driver gives physical memory addresses to the network card when specifying buffer addresses. If the operating system were to change which physical page backs a virtual page used by the driver, the network card would not see the change and write to the wrong page. Thus, the operating system has to be informed of which memory is used for buffers and give it special treatment. While modern hardware can use I/O memory management units to allow devices to address virtual memory, there is a cost to changing I/O memory mappings.

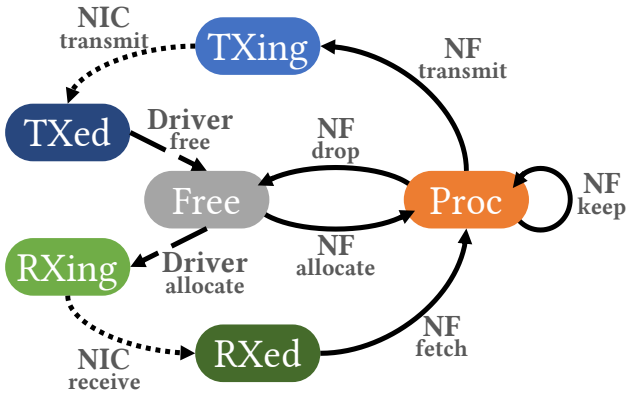


Figure 3. Diagram of the kernel-bypass driver model. Each box is a queue, each arrow is a step moving one packet from one queue to another. Steps are annotated with their actor and their name. "RX" is reception, "TX" is transmission, "Proc" is processing, and "NF" is network function.

This model provides flexibility to network functions: they can keep buffers aside to reassemble messages from high-level protocols such as TCP, and can allocate buffers from the pool in response to non-network events such as timers indicating a request needs to be retried.

The model also lends itself well to concurrency: the "free" queue is the central element shared by any number of reception, transmission or processing queues. A network function can receive and transmit packets from multiple NICs, and it can use multiple processing queues that each communicate with different reception and transmission queues on the same NIC to process packets concurrently and increase overall throughput.

But this flexibility comes at a cost: the steps that the network function can perform besides transmission introduce forks in the path of packet buffers. This requires buffer management within the "free" queue, including support for concurrent accesses. It also requires the driver to implement a policy for buffer freeing and allocation, adding complexity to the overall system.

The model additionally introduces a failure case that is not fundamental to the concept of a network function. If there is a state within the processing logic in which any buffer is kept, and the only way to get out of that state is to receive new data, the system will only make progress if there are buffers outside of the processing queue, which is not guaranteed. Reasoning about the existence of such a state requires reasoning about the invariants that hold in the network function code across packets.

This flexibility is not always needed: some of the network functions that power the backbone of the Internet, such as IP routers or Ethernet bridges, process packets one by one, never keep buffers aside, and never allocate buffers. Overall, they are conceptually simpler than the general case of a network function, yet they must currently pay the price of driver flexibility they do not use.

We propose a new driver model designed for common network functions that do not need the flexibility provided by existing models. It is based on two key insights: we can remove the buffer pool altogether, and we can implement buffer drops on modern NICs without the theoretical branch they introduce, minimizing the amount of state that the driver must keep track of.

Our model is designed to be as simple as possible, thus improving correctness and performance. Its simplicity makes it easier to formally or informally reason about and requires less code and simpler code to implement.

Our model is a subset of the existing model: as shown in Figure 4: the core differences are that it has no pool of free buffers, and does not allow network functions to keep buffers. The driver moves transmitted buffers directly to the reception queue, and the network function must choose to either transmit or drop received packets. This simplifies the driver by giving it only one choice when transmitting a packet: recycling transmitted buffers to the receiving queue now or later. Removing the buffer pool also makes progress easier to reason about: the software can only halt if the driver does not recycle buffers when the receiving queue is empty, or if the network function halts. While termination is impossible to prove in the general case due to the halting problem [31], network functions have strict performance requirements, thus their code is unlikely to have loops whose termination is not obvious because such loops could be performance bugs.

Our model minimizes state by combining reception, processing, and transmission into a single logical descriptor ring containing all buffers, without the need for any other data structure. While it is implemented using one reception ring and one transmission ring, the driver mirrors the head of the transmission ring to the tail of the reception ring, thus ensuring that buffers that have finished transmitting are reused for reception without any intermediate steps.

The key hardware feature that allows this is called "null transmit descriptors": as its name implies, it allows some descriptors in a transmission ring to have no effect. Packet drop is thus a special case of packet transmission, which removes the fork in buffers' paths and allows for a regular buffer flow. For instance, a network card can implement this by dropping packets whose length in metadata is zero.

The driver's job consists of three tasks: move buffers from the "received" queue to the "processing" queue when the network function asks for a packet, move buffers from the "processing" queue to the "transmitting" queue when the network function asks to transmit or drop its current packet, and recycle buffers from the "transmitted" queue to the "receiving" queue to ensure the "receiving" queue is never empty. Since this last operation is not a response to a specific input, the driver must choose when to perform it, for instance once every few transmitted packets.

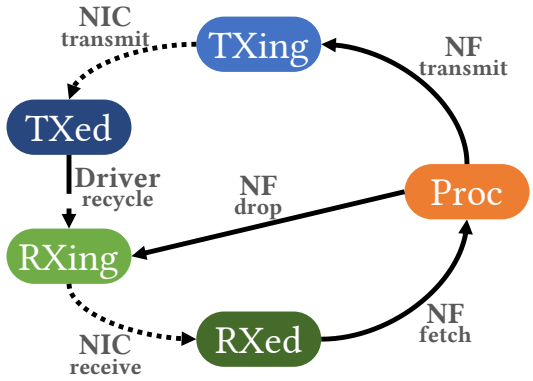


Figure 4. Diagram of our proposed driver model. Semantics are the same as in Figure 3.

Our model supports multiple outputs by using multiple transmission rings and making the driver synchronize their state. That is, the driver must set the tails of all transmission rings at the same time and use the earliest head in all rings as the head to mirror to the reception tail. Transmitting a packet when the driver has multiple outputs conceptually maps to transmitting it on some outputs and dropping it on all others; all rings still have a descriptor pointing to the buffer, but that descriptor is null in some of the rings. This may cause packet drops if an output link is too slow, in which case the entire ring will be used for transmission with no space left for reception. The same could happen in a traditional model if all buffers in the pool were used for transmission due to a slow output.

Multiple inputs can be handled concurrently: while the same processing queue cannot have multiple inputs, since it is not possible to synchronize the state of reception rings, the entire system can be duplicated so that there is one reception queue per input, one associated processing queue, and any number of synchronized transmission queues. Modern NICs have hundreds of queues, thus it is not a problem to use one transmission queue per input.

This does not mean our model requires parallelism: a single thread of execution can implement many instances, which are thus concurrent but not parallel.

Our model is amenable to parallelism: multiple threads of execution can run in parallel, each implementing any number of instances, without having to synchronize any state. Only the state of the rings within an instance needs to be kept in sync. This is similar to existing models.

The key limitation of our model is the flip side of its strength: since network functions must process buffers one by one without keep any aside, they cannot reconstruct multi-packet messages without copying buffers that arrive out of order. Thus, while core functions such as routing and network address translation can be implemented with our model, one cannot terminate TCP connections or otherwise reassemble fragments without copying buffers, which is an expensive operation given modern network speeds.

5. Implementing our new model

In this section, we describe an implementation of our driver model for the Intel 82599 NIC [12] which we call "TinyNF", short for "Tiny Network Function".

TinyNF's goals are to be easy to reason about and fast. The former is different from "correct" because it is hard to tell whether a driver operates as expected without hardware schematics, since the data sheet may be incorrect. However, we want to make it simple enough that it is not a bottleneck in network function verification efforts.

For simplicity, TinyNF processes buffers one at a time: there is always at most one buffer in the processing queue. One key hypothesis in this project was that TinyNF could be fast without explicitly processing packets in batches.

The keys to TinyNF's performance the avoidance of any operation that is not absolutely required, and the use of a few small but surprisingly effective scheduling algorithms for synchronizing queue state.

TinyNF avoids unneeded work, even metadata copy. Because each buffer always belongs to exactly one queue, and because queues are ordered, it is enough to set the buffer pointers at initialization time and never change them afterwards. Moving a buffer from one queue to another only requires writing to the source head and destination tail.

There are fewer delimiters in practice than in theory since some of them are implicit, as shown in Figure 5. The "transmitted" head and tail are the "receiving" tail and "transmitting" head, respectively. Similarly, the "received" head and tail are the "processing" tail and "receiving" head. While there is technically a "processed" queue that does not exist in the conceptual model, its head and tail are the "transmitting" tail and "processing" head respectively. The "processing" tail does not need explicit tracking, because it is always either one buffer ahead of the head or equal to it, due to the one-packet-at-a-time constraint.

TinyNF avoids reading from NIC registers entirely after initialization. To check for received buffers, the "descriptor done" metadata flag of the descriptor at the processing tail is enough. To check for transmitted buffers, the 82599 NIC provides a "transmit head write-back" feature: software can request hardware to write the transmit head to RAM after hardware has finished transmitting a buffer.

TinyNF cannot avoid updating the receive and transmit tails, which are NIC registers and thus slower than RAM, but it can avoid doing so after every packet. Updating the receive tail, which moves buffers to the "receiving" queue, is only necessary once every few transmitted buffers since reception continues working as long as there are buffers in the queue, even if there are less than there theoretically could be. Updating the transmit tail is necessary for buffers to be transmitted to the network, but this can be done once every few transmissions, or when there are no packets to receive and thus no other work to do.

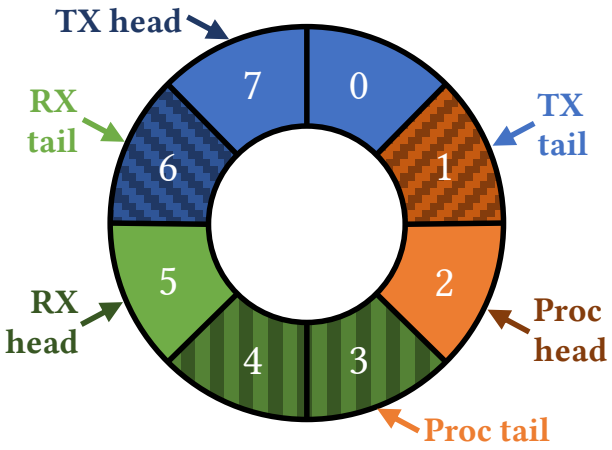


Figure 5. Logical ring composed of reception, processing and transmission queues. "In progress" queues are light, "done" ones are dark and shaded. Heads and tails refer to "in progress" queues, but implicitly delimit the others.

TinyNF carefully schedules operations to minimize the amount of communication between software and hardware. This improves overall latency and reduces the fraction of PCIe throughput used for metadata.

Two operations can be scheduled together: asking the NIC to update the transmission tail and checking for such updates to recycle buffers. The request is made with a bit in transmission metadata, and the check is made by reading the value that the NIC wrote to RAM via DMA. TinyNF schedules both operations once every 64 packets. The check will thus see the update that was requested 64 packets ago.

The most important scheduling decision is updating the transmission tail: frequent updates decrease latency by making the NIC aware of packets sooner, but they increase throughput by performing less book-keeping. Networking stacks such as DPDK solve this with adaptive batching: they check for multiple received buffers at a time up to a limit, let the network function process them all, then update the transmission tail. This theoretically allows drivers to make better scheduling decisions because they have more data: they know how many packets have arrived, rather than whether there is at least one packet.

TinyNF's one-packet-at-a-time model is incompatible with batching, thus we chose an algorithm based on past data instead. TinyNF updates the transmission tail either once every few transmitted packets, or as soon as there are no packets to receive since this likely indicates there is time to perform this expensive operation. This keeps the period short under low load, avoiding latency spikes, but allows for longer periods under high load, avoiding throughput drops, without looking at packets beyond the current one.

Overall, TinyNF is around 550 lines of C code, and its only dependency is a 300-line environment abstraction. It runs entirely in user mode, without kernel dependencies.

6. Evaluation: TinyNF for verification

In this section, we evaluate two hypotheses about TinyNF: its simplicity should (1) make it easier to reason about and (2) make it faster than other verified drivers.

We evaluate TinyNF by using it in the formally verified network functions of the Vigor [33] project. Vigor verifies the entire software stack, including the network function code and the network card driver.

Vigor does not need DPDK's flexibility: it is focused on network functions that form the Internet's backbone, such as Ethernet bridges and IP load balancers. This makes Vigor network functions good candidates to evaluate TinyNF.

Vigor uses DPDK for performance, but it cannot take full advantage of DPDK's optimizations either. Vigor's use of DPDK allows its network functions to outperform those written using traditional networking APIs that go through the kernel to receive and transmit packets. However, some DPDK optimizations such as batching and vectorization are currently out of the reach of automated formal verification. Thus, the driver formally verified by Vigor is the subset of the DPDK driver that can be automatically verified, not all of the driver.

TinyNF makes Vigor network functions 8x faster to verify, as we show in Table 1. We ran verification on two Intel Xeon E5-2690 CPUs at 2.90 GHz, totaling 32 cores.

Vigor verification has two steps: first Vigor symbolically executes the network function code to find all paths, then it validates each path using a theorem prover, which can be done in parallel. Both parts of Vigor verification are faster with TinyNF for the same main reason: symbolic execution does not need to explore DPDK's complex stack, thus it takes $1/5^{\text{th}}$ the time and yields $1/7^{\text{th}}$ the number of paths. Individual paths are also faster to validate since they have less code, though this is less pronounced since validation focuses on network function code, not driver code.

The most drastic change is in the load-balancer, due to its more complex paths that involve more data structures: its total verification time on our machine goes down from ~1h45min to ~14min. This allows full-stack verification to be used as part of development, such as verifying every code change, as opposed to being for special occasions.

	DPDK		TinyNF	
	Sym. ex.	Validation	Sym. ex.	Validation
NAT	337s	149 × 83s	63s	20 × 73s
Bridge	527s	312 × 89s	104s	39 × 77s
LB	731s	297 × 620s	161s	51 × 425s
Policer	392s	190 × 90s	75s	25 × 76s
FW	323s	140 × 83s	61s	20 × 68s

Table 1. Verification time statistics for the Vigor network functions using DPDK and TinyNF.

TinyNF is $1/11^{\text{th}}$ the code of the DPDK driver and has exponentially fewer paths, as we show in Table 2, which explains why the improvements in verification time are so drastic. We measured the code complexity of TinyNF and of the verified subset of DPDK's driver. We manually counted paths, so that we could define them in terms of the public parameters: the arguments passed in the code, and the choices made at DPDK build time when picking a data structure implementation. Automating this using symbolic execution would have only found the number of paths given a concrete configuration. When counting paths, we assume that NIC hardware behaves as per its data sheet.

To show the effect of a change in driver model and not only in implementation, we also included the "Ixy" driver by Emmerich et al. [10], a simplified implementation of DPDK's design for educational purposes that does not aim for comparable performance. As expected, TinyNF and Ixy use similar amounts of code to initialize, since they both use a limited set of NIC hardware features. However, TinyNF has less code and exponentially fewer paths than Ixy in the reception and transmission functions that form the core of the driver, providing more evidence in favor of our model.

We note that the number of paths can change based on programmer decisions: using Boolean expressions rather than conditionally executed code can lower the number of paths, such as writing *x = c ? y : x;* instead of *if (c) { x = y; }* in C. We could have used this to bring down the number of paths in TinyNF's transmission function to 4, without any exponent regardless of the number of output links, but chose not to as such code is compiled to conditional move instructions which have poor tail latency on our machines.

	Init.		Reception			Transmission		
	#funs	*#LoCs*	*#funs*	*#LoCs*	*#paths*	*#funs*	*#LoCs*	*#paths*
DPDK	115	3204	5	136	$1 + A_F + 288A_S$	5	122	$(8 + 14(F_F{}^T + P((F_S + F_F)^T - F_F{}^T))^O$
Ixy	14	279	1	63	$1 + A_F + A_S$	1	53	14^O
TinyNF	4	245	1	17	3	1	29	$2 + 2^O$

A_S, A_F and F_S, F_F: Number of success and failure paths in packet allocation and freeing respectively; Ixy's freeing cannot fail
P: Number of paths in the "put buffers back" operation of the DPDK memory pool in use
T: DPDK parameter for the transmit descriptors write-back threshold, must be >0 O: Number of output links

Table 2. Number of functions, lines of code and paths in DPDK, Ixy and TinyNF drivers for the Intel 82599.

TinyNF makes fewer assumptions on its environment than DPDK. Vigor makes assumptions about the behavior of two components: DPDK data structures and operating system functions.

One fundamental issue with DPDK's driver, even in the verified version, is its need for a data structure to hold free packet buffers. This leaves two options for verification: use a simpler but slower data structure that can be verified or assume that a faster but unverified data structure is correct. Unlike DPDK and TinyNF, there is no evidence that simpler data structures can match their more complex counterparts in performance. In fact, the opposite is true: data structure contracts are already simple yet popular implementations become more complex with time, such as a 2500-line change in Java 8 to make the hash map more resilient to collisions [15]. By comparison, Vigor's verified map has less than 300 lines of code in its entirety.

Another issue with DPDK's driver is in the amount of assumptions it makes about operating system functions. When verifying network functions running on Linux, Vigor replaces these functions during symbolic execution with custom models. This ensures DPDK calls operating system functions correctly according to Linux's documentation, such as by validating the order and arguments of function calls. The models then return symbolic values that cover the range of documented behaviors. But there is no formal specification for these functions, much less a formal proof that the Linux implementation is correct. Thus, Vigor needs to assume the correctness of dozens of models for its proof on Linux. This can be avoided by using a custom operating system, at the cost of losing Linux tools and features such as multitenancy and scheduling. TinyNF needs much less from its environment, drastically reducing the number of assumptions even on Linux.

TinyNF is easier to analyze than DPDK, since it only needs standard C. DPDK uses non-standard extensions to give hints to the CPU and compiler, such as prefetching memory and vectorizing loops. TinyNF does not need any such hints; the driver does not even use the standard library directly, going through a small environment abstraction layer instead.

This standards compliance makes TinyNF analyzable "out of the box" with most tools and allows future tools to support TinyNF without special treatment. This includes symbolic execution engines such as KLEE [4], which Vigor uses and extended to support DPDK code, and manual provers such as VeriFast [14], also used by Vigor. We think this will accelerate networking research in drivers and functions by making it easier to develop new techniques and tools. For instance, TinyNF's simplicity and small size makes it amenable to a proof of functional correctness given a hardware specification, which would improve upon Vigor's proof of memory safety through hardware models.

	DPDK			TinyNF		
	Tput	**Latency** *(µs)*		**Tput**	**Latency** *(µs)*	
	(Gb/s)	*50%*	*99%*	*(Gb/s)*	*50%*	*99%*
NAT	1.99	4.04	4.77	3.69	3.92	4.25
Bridge	2.65	3.97	4.50	5.82	3.93	4.23
LB	2.22	4.01	4.63	6.66	3.90	4.24
Policer	2.96	3.88	4.32	9.53	3.83	4.24
FW	2.65	3.97	4.49	8.14	3.88	4.24

Table 3. Single-link throughput and latency with 1 Gb/s background load of Vigor functions on DPDK and TinyNF.

TinyNF improves the throughput of Vigor network functions by 160%, with 2% less median latency, as we show in Table 3. 99th percentile latency decreases by 7%.

To measure performance, we used two machines in a setup based on RFC 2544 [26], with a "device under test" running a network function and a "tester" running the MoonGen packet generator [9], which can measure latency using NIC timestamps. Both machines run Ubuntu 18.04 on two Intel Xeon E5-2667 v2 CPUs at 3.60GHz with power-saving features disabled and have two Intel 82599ES NICs, using only one port per card to ensure PCIe bandwidth is not a bottleneck. We measure throughput using minimally sized packets. Our workload fills the internal flow table of the network functions to 90% of their capacity. Measuring latency with MoonGen instead of on the device under test allows us to capture the latency of NIC register writes as well as the effects of drivers' NIC configuration. This setup is similar to the one used to originally evaluate Vigor, and can replicate Intel's DPDK performance numbers [7].

We replicate Vigor's benchmark setting: measuring the max throughput that a Vigor network function can achieve with less than 0.1% loss, in a single direction, as well as the latency with 1 Gb/s of background load.

Vigor's NAT gets the lowest throughput improvement; this is because its bottleneck is not the driver but computing packet checksums since it has to modify packet headers. To confirm this, we tried modifying the DPDK version of the NAT to use batching: this results in the same throughput as the TinyNF version of the NAT, confirming that the driver is unlikely to be the bottleneck.

In summary, both of our hypotheses are validated: TinyNF is easier to reason about in terms of code quantity and code complexity, and network functions using TinyNF are faster than the same functions using DPDK's verified subset. Thus, TinyNF allows developers to formally verify their network functions in less time, get more correctness guarantees, more than double the functions' throughput, and lower the functions' median and tail latency.

7. Evaluation: TinyNF in general

In this section, we compare the performance of TinyNF and DPDK for general purpose network functions, regardless of verifiability.

We use the same benchmark setup as in the previous section, but this time use both directions for throughput, for a maximum of 20 Gb/s. We keep throughput symmetric during the benchmarks, i.e., if a function cannot handle a given load, we reduce the load of both directions by the same amount and retry. We then measure the latency at load increments of 1 Gb/s to paint a clear picture of the function's overall performance profile.

TinyNF can outperform a fully optimized DPDK setup, as we show in Figures 6 and 7 using a traffic policer as an example. We compare the Vigor policer using TinyNF as its driver to the same code using either "unbatched" DPDK, which is the simpler version used by Vigor, or "batched" DPDK, which is the standard way to use DPDK that enables optimizations such as adaptive batching and vectorization. We also implemented a 2-core parallelization of the policer for all three variants. We chose the policer because, by design, traffic in one direction is independent of traffic in the other, which means it admits a trivial 2-core parallelization for our experiments. We are not proposing a new way to parallelize network functions, but merely showing that TinyNF can be parallelized in a similar way to existing drivers. This also shows how much improvement parallelization can bring compared to batching.

Using TinyNF, the policer achieves better throughput than using batched DPDK, with an even starker difference when using two cores. The bottleneck that prevents the dual-core TinyNF version of the policer from reaching line rate is the frequent reads from the CPU time, which it needs for flow expiration.

TinyNF leads to better latency at low and high loads but worse latency in the middle, especially the 99th percentile latency. Looking at individual data points, which we show in Figure 8, the TinyNF-based policer has lower latency in some cases, but this advantage is lost in the tail latency. We believe this is a case where DPDK's batching shines: it can detect "gaps" between packets, in which updates to the transmission tail do not compete with packet processing, by looking at how many packets there are in the queue.

Finally, since we had to modify the policer code to use TinyNF, we wanted to see whether the same performance benefits could be obtained without code changes. We wrote a compatibility layer that implements some of the DPDK API on top of TinyNF. The layer cannot implement all of the DPDK API, by design, but can replace DPDK for functions that fit the TinyNF model by changing an environment variable at compile time. The compatibility layer allows for 1% more maximum throughput than batched DPDK, at the cost of increased latency.

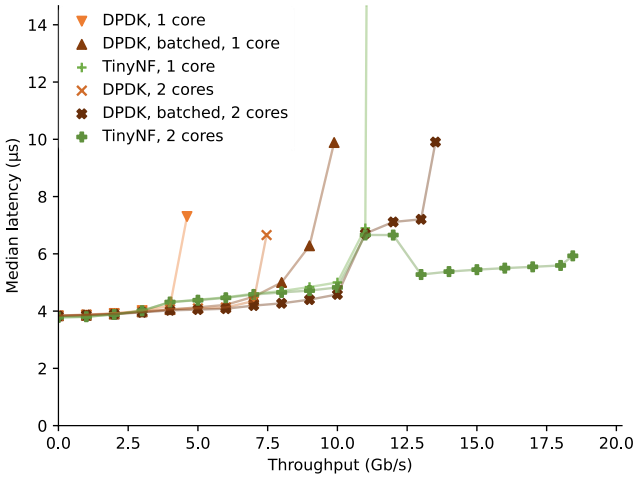


Figure 6. Throughput and median latency of a traffic policer using DPDK with and without batching, TinyNF, and 2-core versions of all three.

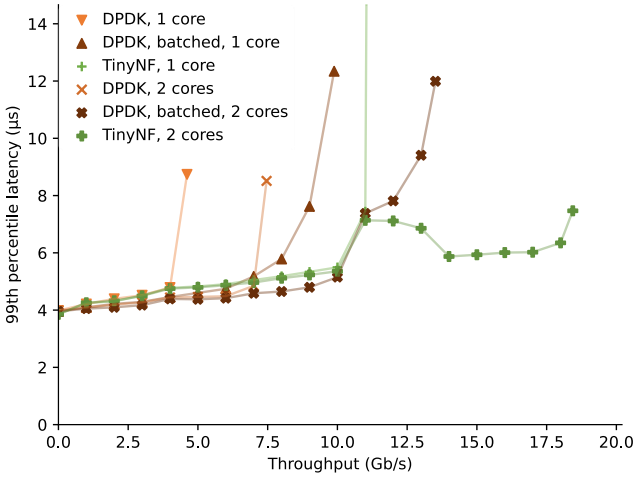


Figure 7. 99th percentile latency version of Figure 6.

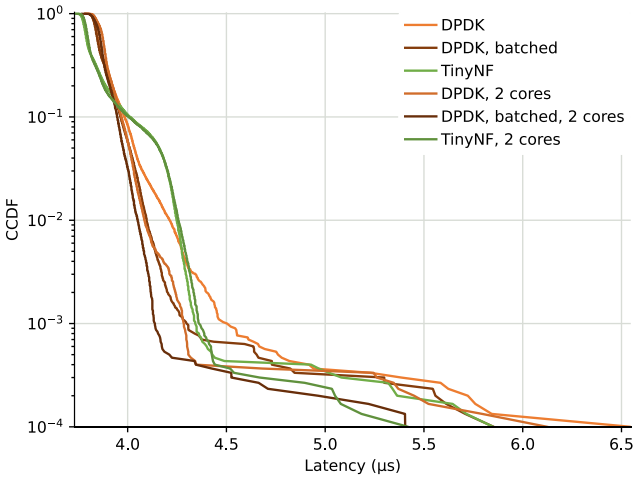


Figure 8. Complementary cumulative latency distributions of a traffic policer using the same alternatives as Figure 6, with 1 Gb/s background load.

A no-op function can handle more throughput with DPDK than with TinyNF, even though the opposite holds with real functions. We reached this surprising conclusion by benchmarking DPDK's "testpmd" built-in application, which DPDK developers use in performance reports [7] to benchmark driver speed. We configured testpmd to update packets' MAC address to provide some realism. Using our setup, both TinyNF and DPDK in its batched mode could saturate two 10 Gb/s links, as we show in Figure 9. We also included the Ixy driver [10], which performed admirably given its educational purpose but could not sustain line rate even with batching.

Since our setup was bottlenecked by link capacity, we chose to lower the CPU frequency to 2 GHz and re-run the benchmark. In this setup, DPDK can reach 97.5% of line rate while TinyNF peaks at around 92.5% of line rate, as we show in Figure 10, though its latency is lower.

We believe the bump around 11 Gb/s is due to hardware issues, since it appears in three independently written drivers and in both a no-op and a nontrivial function.

This result is interesting, since the no-op benchmark is the one used by DPDK developers to measure their progress when optimizing DPDK's performance. If this benchmark does not accurately represent driver performance on real network functions, the DPDK developers may believe they are improving DPDK's performance but do the opposite.

To explain this finding, we started by plotting the no-op function's latency in more detail. We did this because of an observation we made while running the other benchmarks: TinyNF's performance appeared more stable than DPDK's, yielding more consistent results across runs, such as never dropping packets under high loads whereas DPDK would sometimes drop a few packets per million.

As expected, TinyNF has a more stable latency profile than DPDK: without background load, TinyNF's latency remains low up until the 99.9th percentile, whereas DPDK's latency starts jittering before this, as show in Figure 11. We stop at the 99.99th percentile because Primorac et al. showed that NIC timestamping is not accurate after that point [23].

This measurement highlights a key issue with DPDK's driver model: the driver has to manage buffers explicitly instead of merely moving them from one queue to the next, which leads to a distinct bump in latency before the 99th percentile. The same holds for Ixy, since it uses the same driver model as DPDK.

We used the toplev microarchitectural measurement tool [22] to investigate bottlenecks in DPDK's driver when running the Vigor policer. While the tool indicates that the policer is bottlenecked on memory writes, there is no single write that dominates. Some of the memory writes that take the most time are fundamental to DPDK's design, such as moving buffer pointers to and from the buffer pool, while others could be removed at the cost of some functionality, such as writes to packet buffer metadata.

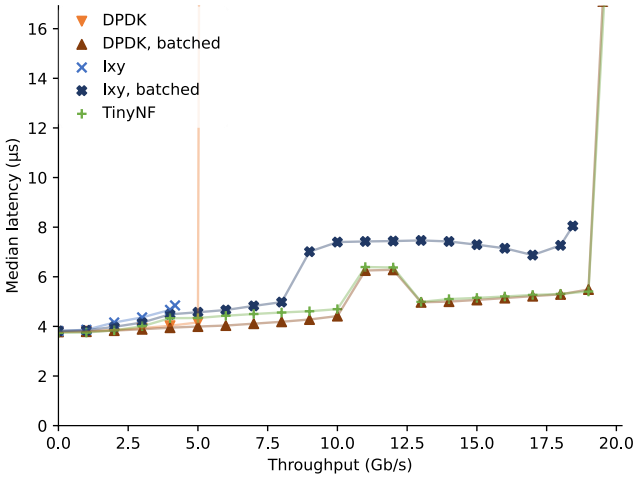


Figure 9. Throughput and median latency of DPDK's no-op function with and without batching, a port of it on TinyNF, and a port of it on Ixy with and without batching.

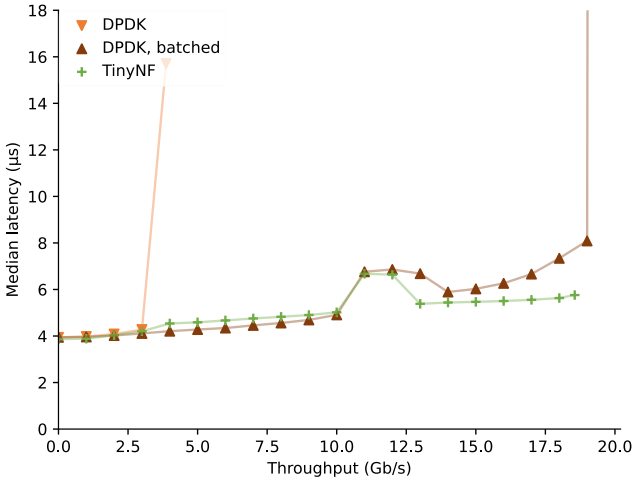


Figure 10. Same benchmark as Figure 9 but with the CPU capped to 2 GHz. We do not show Ixy since it could not sustain line rate even at full CPU speed.

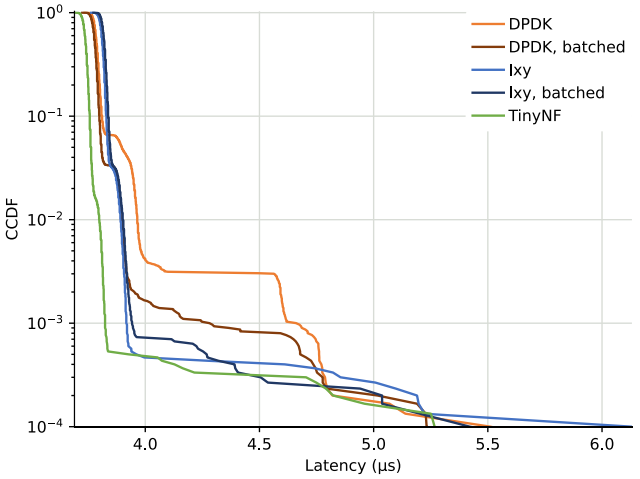


Figure 11. Complementary cumulative latency distributions of the no-ops from Figure 9 without background load.

TinyNF slows down less when running real functions because its instruction-level parallelism has room to grow and it interferes less with the CPU's caches. We reached this conclusion after measuring low-level CPU counters using libPAPI [30], in particular the number of cycles, instructions, and cache hits per packet at 20 Gb/s.

Before going further, we must caution against over-interpreting our results, in particular absolute numbers of cycles. To measure low-level CPU metrics, we instrumented network functions with code that copies counter values for later processing. This has overhead: reading performance counters uses cycles, and copying their values touches the CPU caches. Furthermore, due to the out-of-order nature of modern CPUs, accurately measuring cycle counts requires inserting serializing instructions to ensure past instructions have completed. Thus, measuring the cycle count increases it as it prevents the CPU from reordering some instructions. The measurement overhead is stable, so we measure it and subtract it from the measurements, but we cannot fully account for cache changes due to storing counter values, or for the effects of serialization. Because of this, cycle counts can only be compared to other functions on the same driver. Instruction counts and cache use can be compared globally.

We collected data by running network functions ten times collecting ten million packets each time. We intended to collect data in a single run, but noticed that some runs have a lower cache miss rate than others, despite using the same executable run in the same way on a CPU not otherwise used by the operating system.

We used four functions: a no-op function that does not even touch packets, one that writes a constant to the destination MAC address, one that sets the destination MAC address using a lookup table based on the source MAC address, and the Vigor policer. In our setup, the write function is faster on DPDK but the lookup one is faster on TinyNF. We report the measured cycles, instructions and cache hits in Table 4. We do not report main memory hits as they are negligible, around one in a million packets.

Two results stand out: the increase in instructions per cycle for TinyNF when running more realistic functions, and TinyNF's low cache use compared to DPDK.

TinyNF has low instruction-level paralelism in a no-op because the CPU is waiting for operations on descriptors and NIC registers, which cannot be executed out of order. On a more realistic function, the CPU executes the function instructions out of order, increasing efficiency, thus the slowdown is not linear. This is consistent with TinyNF's low latency in the reduced frequency benchmark: the frequency makes little difference when waiting for the NIC.

Batched DPDK, on the other hand, can execute multiple instructions per cycle even in no-ops, due to instructions for metadata and buffer management. Its use of vector instructions also helps keep a high instruction count per cycle by waiting for multiple descriptors in parallel without reordering. The slowdown when executing a real function is thus linear in the number of instructions, unlike TinyNF.

TinyNF also has a lower memory footprint than DPDK, thus realistic functions have fewer cache misses, an effect that cannot be observed in no-ops.

	IPC	Cycles		Instrs		L1d hits		L2 hits		L3 hits	
	50%	*50%*	*99%*	*50%*	*99%*	*50%*	*99%*	*50%*	*99%*	*50%*	*99%*
DPDK unbatched											
No-op	0.39	664	2140	258	3780	101	1300	8.94	103	1.00	82.0
MAC write	0.37	725	2220	267	3790	107	1300	10.3	102	2.00	85.0
MAC lookup	0.39	746	2180	287	3810	116	1310	10.4	96.1	3.00	96.0
Policer	0.66	866	2540	669	4130	331	1500	4.94	94.4	3.00	95.0
DPDK batched											
No-op	1.70	58.1	64.3	99.0	99.1	32.3	33.0	4.81	5.83	1.41	2.50
MAC write	1.68	63.9	70.1	107	107	36.3	37.0	4.74	5.65	2.66	3.62
MAC lookup	1.53	84.4	93.1	129	129	46.6	47.3	5.01	6.07	5.12	5.94
Policer	1.65	298	333	511	512	265	269	4.33	5.52	4.47	5.53
TinyNF											
No-op	0.12	289	683	35.0	53.0	7.87	16.7	4.51	11.0	0.00	1.00
MAC write	0.13	339	717	45.0	63.0	13.8	22.2	5.11	12.8	1.00	3.00
MAC lookup	0.18	360	734	65.0	83.0	19.7	29.7	8.99	14.9	2.00	4.00
Policer	0.49	490	883	297	308	125	144	11.0	23.0	2.00	4.00

Table 4. Low-level metrics. IPC is Instructions Per Cycle. Cycles and IPC are only comparable within the same driver, as explained in the main text. DPDK batched uses batches of size 32. Main memory hits are negligible and not shown.

8. Applicability

In this section, we evaluate the applicability of our model to real-world network function deployment. Did we strike a good tradeoff choosing not to support some functions to simplify the model? And is our model useful in the context of network function virtualization?

As previously explained, the core limitation of our driver model is that network functions cannot keep buffers aside for later use. For instance, they cannot reconstruct messages in TCP or other higher-level protocols. Our model targets network functions that do not need to do so because they logically handle packets one at a time.

Our model supports many well-known functions, though there is no standard list of network functions. Despite their increased importance in modern networking, there is no consensus on what is a "network function" and what is not. There have been attempts such as RFC 3234 [27] to classify "middleboxes", which are functions that are not crucial to the network, but to the best of our knowledge there is no commonly accepted list of network functions. We chose to use the list of functions from the ClickOS [18] paper, which were also used by the authors of Vigor [33] to estimate the applicability of their verification technique. We complement this list with our own knowledge, for lack of a more standard source.

Our driver model supports 13 of the 14 types of network functions listed in ClickOS: load balancing, DPI, NAT, firewalls, tunnel, multicast, BRAS, monitoring, DDoS prevention, IP proxies, congestion control, IDS, and IPS. The only one that our model cannot support without compromises is a traffic shaper, because shaping requires keeping packets to send them later in the desired traffic shape. Among the network functions not mentioned by ClickOS, our model can be used for Ethernet bridges, ARP clients and servers, DNS proxies, statistics collectors, traffic policers, and Google's Maglev [8] load-balancer.

However, our driver model cannot efficiently support functions based on entire TCP messages, since this requires keeping IP packets around to reorder and merge them into logical messages. Such functions include proxies and HTTP servers. While one could implement reordering by copying buffers before giving descriptors back to the hardware, this would hinder performance.

We believe our model is a good fit for network functions that form the backbone of networks, such as routing, load-balancing, NAT and DNS, access control and statistics. However, it is not suited to high-level functions that deal with entire connections or protocols that fragment packets.

Some requirements are orthogonal to our model. For instance, offloading checksums to hardware would remove the main bottleneck in the NAT we benchmarked. Any such feature that can be used by providing metadata to the NIC can be implemented in a driver using our model.

TinyNF can be used for virtualization, which is a key tool for the practical deployment of network functions [35]. Virtualization allows operators to deploy multiple network functions on the same physical machine, instead of having to dedicate an entire machine to a single function. They also provide an easier way to manage network functions, in the same way virtual machines ease software management.

We experimented with virtualization using Single-Root I/O Virtualization, or "SR-IOV" for short, a PCIe standard with which network cards can expose virtual network cards with the same packet-processing features as the physical card. The virtual machine monitor can let virtual machines access virtual devices directly, without surrendering control over the physical card. The physical card includes hardware to route packets to virtual cards based on packet headers, for instance by Ethernet address. The physical card can limit the rate at which each virtual card transmits packets and can prevent virtual cards from transmitting packets with a different source address than their own. Virtual machines thus gain the benefits of direct access without the ability to monopolize the link or lie about their network identity.

The Intel 82599's virtual cards do not support some of the physical features. Notably, using transmit head write-back causes virtual cards to hang, a problem not mentioned in the card's data sheet but already reported by the authors of Arrakis [20]. Another missing feature is legacy packet descriptors, which are simpler to use, though the data sheet calls this out. We wrote a version of TinyNF that does not use these features, making it slightly slower. The Arrakis authors estimated that the lack of transmit head write-back causes a 5% performance penalty

We used the same physical setup as before, but with 16 virtual functions on each of the two network cards, for a total of 32 virtual cards. Each virtual card has an Ethernet address, and physical cards route packets to virtual cards based on these addresses. The only code changes are due to the missing features mentioned above, as well as a few dozen lines of configuration. The functions forward each packet using a virtual card on the physical card opposite the one whose virtual card received the packet.

The Vigor policer handles 12.2 Gb/s of minimally-sized packets without loss when using TinyNF in this setup. A no-op function reaches 14 Gb/s. Both are bottlenecked by reading packet descriptors for packet fetches, as the data from packets and descriptors no longer fits in the L2 cache.

This experiment is only intended to show that our driver model is applicable to virtualized environments. With this number of devices, other concerns arise such as load skew across devices and non-uniform memory accesses, which we do not capture here. We believe TinyNF is as sensitive to these concerns as other stacks. In particular, the order in which the function checks virtual cards for packets matters. For instance, if packets mostly arrive on one card, checking the other cards for packets will limit performance.

9. Discussion

In this section, we present our main takeaways from this project, in the form of actionable recommendations for both researchers and practitioners.

Drivers are not a special category of software, and the line currently drawn between drivers and other kinds of software is neither well-defined nor helpful. Drivers should be considered just another kind of software system, one that is more focused on hardware than usual. The same techniques used in systems that handle requests can and should be scaled down to "drivers", instead of creating new vocabulary for one kind of software.

The common meaning of "driver" is a piece of code that has exclusive access to hardware and exposes a software API to programs who want to use it. However, software that does this is not always called a driver. Operating systems allows programs to access CPUs, including isolation and high-level APIs to access features such as clocks, but they are not commonly referred to as "CPU drivers", with the notable exception of Barrelfish [1]. The same can be said of higher-level frameworks such as Java or .NET, which offer an abstraction over low-level CPU details yet are not called drivers. This applies to other kinds of devices as well: code that lets programs run GPU shaders is called a driver, but code that lets programs to draw windows and buttons on the screen is not, even though it is also a way for programs to draw. The internal architecture of some systems does rely on "drivers" as an indirection to access hardware, but this is not relevant from users' point of view.

An example of overly specific vocabulary is "batching" in network drivers: a feature that improves performance by amortizing costs. It is really composed of three independent features: (1) getting multiple packets at a time from the NIC, gaining information about network load, (2) processing multiple packets at a time, allowing for vectorized code, and (3) giving multiple packets at a time to the NIC, amortizing the cost of NIC register writes. TinyNF shows that only (3) is required for high throughput, though (1) may be required to get consistently low latency. In fact, any developer that uses batching but does not explicitly keep track of network load or use vector operations is already implicitly aware of this. Amortizing NIC writes is similar to existing techniques such as buffering reads and coalescing writes in disk I/O.

The idea that drivers are a special kind of software is hindering research. Most systems for fast networking, such as ClickOS [18], DPDK [5], netmap [28], SoftNIC [11], and IX [2], reuse existing drivers, which are bottlenecks on their performance. Arrakis [20] uses custom drivers but focuses on interrupt-driven I/O, which strikes a different tradeoff. Ixy [10], is the only research driver we know of besides ours. It is odd to have more research operating systems than drivers: the former are by definition more complex as they contain at least one driver.

Isolation is required for low-level performance, just as modularity is required for high-level correctness. The best-effort approach of shared caches is no longer enough when interferences that cause even a low number of cache misses cause a noticeable performance difference, as is the case with fast networking.

One way to provide performance modularity is to run each part of a system on physically separate hardware, as in TAS [16]. This eliminates interference in per-core caches, at the cost of increasing resource use. It also increases the cost of communication between modules, in the same way protection rings eliminate functional interference between user and kernel mode at the cost of an expensive boundary between the two modes.

However, the current way to measure low-level metrics through special CPU registers cannot be isolated from the code under measurement. This is not an issue for most code, because the overhead of measurement is low, but it becomes an issue with nanosecond-scale code such as TinyNF.

One way to avoid measurement overhead is to use static instead of dynamic analysis, but this requires a hardware model. TiML [32] includes performance reasoning in a type system, and Bolt [13] infers performance metrics from the source code of network functions written in C. However, predicting cycle counts requires accurate hardware models. For instance, Bolt predicts instruction counts within a few percent of ground truth but is 300% off the true cycle count for typical workloads. Since hardware optimizations are considered a competitive advantage, perfectly accurate hardware models are unlikely to be made publicly available.

Standard benchmarks would improve the state of network function research. Other areas of research use benchmarks such as SPEC [3] to measure improvements on a widely-accepted scale. There is no equivalent for network functions, not even non-standard ones.

We chose to explore a new point in the design space of networking code based on our experience with networking research, but the main threat to this paper's validity is that we have no way to validate the usefulness of this design. It may be that real-world traffic looks more like the one used to benchmark Arrakis [20], for instance, in which Peter et al. came to the conclusion that handling operations in user mode entirely eliminates the need for even transmission tail update coalescing.

Unlike other domains in which one can substitute benchmarks with well-known publicly available targets, such as compiling the compiler itself to show optimization improvements, network functions are generally not public.

This problem is getting worse as hardware gets faster. With 100 Gb/s Ethernet becoming more popular, should we focus on handling minimally sized packets, with a budget of 6ns per packet, or should we assume that traffic is made up of packets in the hundreds of bytes, as Pigasus [34] does? We do not know.

Any benchmark, even if unrealistic, would improve the situation, which is that both industry and academia use no-op functions as a de facto standard. DPDK's performance reports [6] from Intel, Mellanox, and Broadcom all exclusively use no-ops, and research such as netmap [28] or SoftNIC [11] mostly use no-ops. But no-ops are not representative of either general or specific cases. Even overly specific benchmark suite would at least result in systems optimized for a real use case, instead of systems optimized for no-ops which are not useful to anyone.

Since the performance of infrastructure code interferes with the performance of application code in non-obvious ways, extrapolations from no-ops are not representative of actual performance. We believe that using any real network function as a standard benchmark would provide a data point from which one can extrapolate more credibly to other real functions. The chosen function would be closer to any other function than a no-op is in terms of how the infrastructure code influences its performance, regardless of how close it is in terms of functionality.

We started this project with the goal to close the gap between unverified and verified performance using the Vigor network functions as benchmarks. Had we measured no-op performance for TinyNF first, under the belief that it was representative, we would have come to the conclusion that it was worse than DPDK. This could have led us to make TinyNF more complex to "fix" its no-op performance, accidentally lowering performance for real functions in the process.

More formal hardware data sheets could speed up software development and reduce bugs, without the need to change the hardware. TinyNF's complexity mainly comes from the number of assumptions it makes about hardware. These are due to missing or incorrect data, which is a natural consequence of free-form data sheets.

Most of the data sheet errors could be avoided using the same kind of analysis performed by compilers today. For instance, the Intel 82599 NIC's data sheet [12] has typos in register names and even in the size of some register fields; these could be caught by consistency checks ensuring all referenced names are declared and all registers contain the right number of bits. Some registers are only documented within the list of registers and not in the explanations of the operations they are used for, requiring developers to read the entire data sheet to learn about them; these could be caught by a check for unused declarations.

It would be unreasonable to expect hardware engineers to always provide perfect data sheets or design bug-free hardware, in the same way that it would be unreasonable to expect software engineers to always write bug-free code. However, our experience is that most current bugs are low-hanging fruit that could be caught without inventing new analysis techniques, if data sheets written in a machine-readable format first.

Using the basic features of a modern NIC does not have to be complicated, despite the belief that hardware has become inherently harder to deal with than in the past. We examined the oldest driver we could find for a NIC of the Intel 8259x family, which is the so-called "apricot" driver [17] for the Intel 82596, released with Linux 1.1 in 1994. It contains 450 lines of code not including debug code, which is close to TinyNF's 550.

Most lines of code in TinyNF come from unused features that must be initialized anyway. For instance, software must clear packet filters and virtualization-related registers after resetting the hardware, unlike some other features that are left in a clean state by the hardware reset. This kind of issues is not a fundamental source of complexity but a hardware implementation detail. If the hardware could be fully reset in a single operation, TinyNF would have fewer lines of code than the old "apricot" driver. This overhead is not as visible in a driver such as DPDK's, whose complexity comes from the amount of features it supports.

We hope this paper serves as evidence that developing code that interacts with network cards is both interesting and rewarding, and that it is not as complex or difficult as is often believed. On the contrary, we found that developing our own driver made the development and verification of network functions easier, by removing all dependencies on complex external stacks and kernel-mode drivers.

Acknowledgements

We thank our shepherd Simon Peter for his useful feedback and guidance; the anonymous reviewers for their useful and detailed reviews; the anonymous artifact evaluators for their useful feedback on the code and experiments; Arseniy Zaostrovnykh, Akvilė Valentukonytė, Blagovesta Kostova, Katerina Argyraki, Lei Yan, Rishabh Iyer, Samuel Chassot, and Yassmine Abdrabo, for providing feedback on the ideas, paper, and code.

Availability

Our code is available at *github.com/dslab-epfl/tinynf*, and described further in the Artifact Appendix below. The code obtained the "Artifact Available", "Artifact Functional" and "Results Reproduced" badge from artifact evaluation and can thus be reused by others with confidence.

In particular, the TinyNF code can be used as a simpler and faster base for any network function that fits its model, or as a baseline to evaluate low-level networking code. The benchmarking scripts are independent of TinyNF and can be reused to measure the performance of network functions that use any framework or driver.

References

[1] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), 29–44.

[2] Belay, A., Prekas, G., Primorac, M., Klimovic, A., Grossman, S., Kozyrakis, C. and Bugnion, E. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4 (Dec. 2016). DOI:https://doi.org/10.1145/2997641.

[3] Bucek, J., Lange, K.-D. and v. Kistowski, J. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2018), 41–42.

[4] Cadar, C., Dunbar, D. and Engler, D. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), 209–224.

[5] Data Plane Development Kit: *https://www.dpdk.org/*. Accessed: 2020-01-21.

[6] DPDK - Performance reports: *http://core.dpdk.org/perf-reports/*. Accessed: 2020-05-26.

[7] DPDK Intel NIC Performance Report Release 20.02: *https://fast.dpdk.org/doc/perf/DPDK_20_02_Intel_NIC_performance_report.pdf*. Accessed: 2020-05-27.

[8] Eisenbud, D.E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W. and Hosein, J.D. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), 523–535.

[9] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F. and Carle, G. 2015. MoonGen: A Scriptable High-Speed Packet Generator. *Proceedings of the 2015 Internet Measurement Conference* (New York, NY, USA, 2015), 275–287.

[10] Emmerich, P., Pudelko, M., Bauer, S., Huber, S., Zwickl, T. and Carle, G. 2019. User Space Network Drivers. *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (Los Alamitos, CA, USA, Sep. 2019), 1–12.

[11] Han, S., Jang, K., Panda, A., Palkar, S., Han, D. and Ratnasamy, S. 2015. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report #UCB/EECS-2015-155. EECS Department, University of California, Berkeley.

[12] Intel 82599 10 Gigabit Ethernet Controller Technical Library: *https://www.intel.com/content/www/us/en/design/products-and-solutions/networking-and-io/82599-10-gigabit-ethernet-controller/technical-library.html*. Accessed: 2020-01-21.

[13] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K. and Candea, G. 2019. Performance Contracts for Software Network Functions. *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), 517–530.

[14] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W. and Piessens, F. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *Proceedings of the Third International Conference on NASA Formal Methods* (Berlin, Heidelberg, 2011), 41–55.

[15] JDK-8023463: Improvements to HashMap / LinkedHashMap use of bins/buckets and trees: *https://bugs.openjdk.java.net/browse/JDK-8023463*. Accessed: 2020-09-08.

[16] Kaufmann, A., Stamler, T., Peter, S., Sharma, N.Kr., Krishnamurthy, A. and Anderson, T. 2019. TAS: TCP Acceleration as an OS Service. *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019).

[17] Linux 1.1.23. The "apricot" driver is in drivers/net/apricot.c.: *https://mirrors.edge.kernel.org/pub/linux/kernel/v1.1/*. Accessed: 2020-01-21.

[18] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R. and Huici, F. 2014. ClickOS and the Art of Network Function Virtualization. *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (USA, 2014), 459–473.

[19] Neugebauer, R., Antichi, G., Zazo, J.F., Audzevich, Y., López-Buedo, S. and Moore, A.W. 2018. Understanding PCIe Performance for End Host Networking. *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), 327–341.

[20] Peter, S., Li, J., Zhang, I., Ports, D.R.K., Woos, D., Krishnamurthy, A., Anderson, T. and Roscoe, T. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.* 33, 4 (Nov. 2015). DOI:https://doi.org/10.1145/2812806.

[21] Pirelli, S., Zaostrovnykh, A. and Candea, G. 2018. A Formally Verified NAT Stack. *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks, KBNets@SIGCOMM 2018, Budapest, Hungary, August 20, 2018* (2018), 8–14.

[22] pmu-tools GitHub repository: *https://github.com/andikleen/pmu-tools*. Accessed: 2020-09-11.

[23] Primorac, M., Bugnion, E. and Argyraki, K. 2017. How to Measure the Killer Microsecond. *Proceedings of the Workshop on Kernel-Bypass Networks* (New York, NY, USA, 2017), 37–42.

[24] Registered Input/Output (RIO) API Extensions: *https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh997032(v=ws.11)*. Accessed: 2020-01-21.

[25] Renzelmann, M.J., Kadav, A. and Swift, M.M. 2012. SymDrive: Testing Drivers without Devices. *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), 279–292.

[26] RFC 2544 - Benchmarking Methodology for Network Interconnect Devices: 1999. *https://www.ietf.org/rfc/rfc2544.txt*. Accessed: 2020-05-26.

[27] RFC 3234 - Middleboxes: Taxonomy and Issues: *https://tools.ietf.org/html/rfc3234*. Accessed: 2020-01-21.

[28] Rizzo, L. 2012. Netmap: A Novel Framework for Fast Packet I/O. *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (USA, 2012), 9.

[29] sys/socket.h - main sockets header: *https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sys_socket.h.html*.

[30] Terpstra, D., Jagode, H., You, H. and Dongarra, J. 2010. Collecting Performance Data with PAPI-C. *Tools for High Performance Computing 2009* (Berlin, Heidelberg, 2010), 157–173.

[31] Turing, A.M. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*. s2-42, 1 (1937), 230–265. DOI:https://doi.org/10.1112/plms/s2-42.1.230.

[32] Wang, P., Wang, D. and Chlipala, A. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017). DOI:https://doi.org/10.1145/3133903.

[33] Zaostrovnykh, A., Pirelli, S., Iyer, R., Rizzo, M., Pedrosa, L., Argyraki, K. and Candea, G. 2019. Verifying Software Network Functions with No Verification Expertise. *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), 275–290.

[34] Zhao, Z., Sadok, H., Atre, N., Hoe, J., Sekar, V. and Sherry, J. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2020).

[35] 2012. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. Issue 1. ETSI.

Artifact Appendix

Abstract

The artifact of this paper contains the code of the "TinyNF" prototype, as well as scripts to run the various experiments used in this paper. The benchmarking scripts can be used for any network function, not only TinyNF.

Checklist

Program: driver and network functions
Metrics: throughput, latency, code complexity
Output: compiled network function including the driver
Experiments: as described in Sections 6, 7, and 8
Required disk space: 80 GB for low-level metrics, a few MBs for everything else
Expected run time: around half a day to run all available experiments, almost all of which is spent waiting
Public link: *github.com/dslab-epfl/tinynf*
Code license: MIT

Description

How to access: Use the link above.
Hardware dependencies: Two machines with Intel 82599 NICs, as mentioned in *experiments/ReadMe.md*.
Software dependencies: TinyNF currently supports Linux only. A few standard packages are required to compile and run experiments, as described in *experiments/ReadMe.md*.

Installation

There is no explicit installation step, cloning the repository is enough. The artifact is fully self-contained and does not install files to the rest of the machine, except for benchmark scripts copied to a configurable directory on the machine that runs them.

Experiment workflow

All experiments are run using scripts. Manual work is not needed beyond executing the scripts with some parameters and setting up a configuration file once.

Evaluation and expected result

The scripts produce tables and figures that correspond to those in this paper. Tables are produced as tab-separated output on the command line, while figures are produced as vector images.

Experiment customization

The benchmarking scripts are reusable for any experiment even not including TinyNF. They are designed to measure the throughput and latency of any network function, with special treatment for ones that require DPDK-compatible kernel drivers.

Notes

We elaborate here on the environment abstraction library mentioned in Section 5, which is the only dependency of the TinyNF driver. The driver itself does not depend on any kernel-mode driver and only needs "freestanding" features of the C library, i.e., it only uses a few headers and types but no C functions from the standard library.

The abstraction contains 5 groups of functions: memory allocation and deallocation, translation between virtual and physical addresses, PCI register reads and writes, endianness conversion, and sleep.

We believe these 5 groups are all necessary to write NIC drivers without compromises, though some of these could be modified or removed under certain conditions. Sleeping could be replaced by a clock function combined with busy-waiting in the driver, but this would be less efficient and no less complex. Memory deallocation could be omitted if the software uses a crash-only failure mode. Translating virtual to physical addresses may not be necessary in the presence of an IOMMU, if memory allocation also configured the IOMMU. In systems that use the Enhanced Configuration Access Mechanism for PCI registers, or "ECAM" for short, the functions to read and write PCI registers could instead be a single function providing the memory address at which this space is accessible for a given device.

Notably, the abstraction does not expose non-uniform memory access: implementations are expected to provide sane defaults. The current Linux implementation allocates memory on the same node as the current CPU and does not allow for PCI operations on devices on other nodes. This would need to change in a more production-ready version, in which various strategies can be used when dealing with devices on multiple nodes, but those strategies are beyond the scope of this paper.

AE Methodology

Submission, reviewing and badging methodology:
usenix.org/conference/osdi20/call-for-artifacts



PANIC: A High-Performance Programmable NIC for Multi-tenant Networks

Jiaxin Lin
University of Wisconsin-Madison

Kiran Patel
University of Illinois at Chicago

Brent E. Stephens
University of Illinois at Chicago

Anirudh Sivaraman
New York University (NYU)

Aditya Akella
University of Wisconsin-Madison

Abstract

Programmable NICs have diverse uses, and there is a need for a NIC platform that can offload computation from multiple co-resident applications to many different types of substrates, including hardware accelerators, embedded FPGAs, and embedded processor cores. Unfortunately, there is no existing NIC design that can simultaneously support a large number of diverse offloads while ensuring high throughput/low latency, multi-tenant isolation, flexible offload chaining, and support for offloads with variable performance.

This paper presents PANIC, a new programmable NIC. There are two new key components of the PANIC design that enable it to overcome the limitations of existing NICs: 1) A high-performance switching interconnect that scalably connects independent engines into offload chains, and 2) A new hybrid push/pull packet scheduler that provides cross-tenant performance isolation and low-latency load-balancing across parallel offload engines. From experiments performed on an 100 Gbps FPGA-based prototype, we find that this design overcomes the limitations of state-of-the-art programmable NICs.

1 Introduction

The gap between network line-rates and the rate at which a CPU can produce and consume data is widening rapidly [71, 66]. Emerging *programmable ("smart") NICs* can help overcome this problem [32]. There are many different types of offloads that can be implemented on a programmable NIC. These offloads, which accelerate computation across all of the different layers of the network stack, can reduce load on the general purpose CPU, reduce latency, and increase throughput [32, 48, 59, 69, 13].

Many different cloud and datacenter applications and use cases have been shown to benefit from offloading computation to programmable NICs [13, 48, 59, 42, 32, 37, 49, 46, 62, 47, 30, 36, 70, 69, 35, 55, 45]. However, there is no single "silver bullet" offload that can improve performance in all cases. Instead, we anticipate that different applications will specify their own chains of offloads, and that the operator will then merge these chains with infrastructure-related offloads and run them on her programmable NICs. To realize this vision, this paper presents PANIC, a new scalable and high-performance programmable NIC for multi-tenant networks that supports a wide variety of different types of offloads and composes them into isolated offload chains.

To enable cloud operators to provide NIC offload chains as a service to tenants, a programmable NIC must support: 1) Offload variety: some offloads like cryptography are best suited for hardware implementations, while an offload providing a low-latency bypass for RPCs in an application is better suited for an embedded core [51]; 2) Offload chaining: to minimize wasted chip area on redundant functions, the NIC should facilitate composing independent hardware offload units into a chain as needed, with commonly-needed offloads shared across tenants; 3) Multi-tenant isolation: tenants should not be able to consume more than their allocation of a shared offload; 4) Variable-performance offloads: there are useful offloads that are not guaranteed to run at line-rate, as well as important offloads that run with low latency and at line-rate.

There exist many different programmable NICs [32, 12, 75, 31, 58, 72, 23, 24, 11, 57, 53, 54, 52, 76], but, there is no programmable NIC that is currently able to provide all of the above properties. Existing NIC designs can be categorized as follows, with each category imposing key limitations:

- **Pipeline-of-Offloads NICs** place multiple offloads in a pipeline to enable packets to be processed by a chain of functions [52, 32]. Chaining can be modified in these NICs today but requires a significant amount of time and developer effort for FPGA synthesis, and slow offloads cause packet loss or head-of-line (HOL) blocking.
- **Manycore NICs** load balance packets across many embedded CPU cores, with the CPU core then controlling the processing of packets as needed for different offloads [23, 24, 53, 54, 57, 72, 58]. These designs suffer from performance issues because embedded CPU cores add tens of microseconds of additional latency [32]. Also, no existing manycore NICs provide performant mechanisms to isolate competing tenants. Further, performance on manycore NICs can degrade significantly if the working set does not fit within the core's cache.
- **RMT NICs** use on-NIC reconfigurable match+action (RMT) pipeline to implement NIC offloads. The types of offloads that can be supported by RMT pipelines are limited because each pipeline stage must be able to handle processing a new packet every single clock cycle.

This paper presents the design, implementation and evaluation of PANIC, a new NIC that overcomes the key limitations of existing NIC designs. PANIC draws inspiration from recent work on reconfigurable (RMT) switches [21, 67, 68, 27, 16]. PANIC's design leverages three key principles:

1. *Offloads should be self-contained.* The set of potentially useful offloads is diverse and vast, spanning all of the layers of the network stack. As such, a programmable NIC should be able to support both hardware IP cores and embedded CPUs as offloads.
2. *Packet scheduling, buffering, and load-balancing should be centralized* for the best performance and efficiency because decentralized decisions and per-offload queuing can lead to poor tail response latencies and poor buffer utilization due to load imbalances.
3. Because the cost of small/medium-sized non-blocking fabrics is small relative to the NIC overall, *the offloads should be connected by a non-blocking/low-oversubscribed switching fabric* to enable flexible chaining of offloads.

Following these design principles, this paper makes three key contributions: 1) A novel programmable NIC design where diverse offloads are connected to a non-blocking switching fabric, with chains orchestrated by a programmable RMT pipeline, 2) A new hybrid push/pull scheduler-and-load balancer with priority-aware packet dropping, and 3) An analysis of the costs of on-NIC programmable switching and scheduling that finds them to be low relative to the NIC as a whole.

The PANIC NIC has four components: 1) an RMT switch pipeline, 2) a switching fabric, 3) a central scheduler, and 4) self-contained compute units. The RMT pipeline provides programmable chain orchestration. A high performance interconnect enables programmable chaining at line-rate. The central scheduler provides isolation, buffer management, and load-balancing. Self-contained compute units may be either hardware accelerators or embedded cores and are not required to run at line-rate.

To evaluate the feasibility of PANIC, we have performed both ASIC analysis and experiments with an FPGA prototype. Our ASIC analysis demonstrates the feasibility of the PANIC architecture and shows that the crossbar interconnect topology scales well up to 32 total attached compute units. Our FPGA prototype can perform dynamic offload chaining at 100 Gbps, and achieves nanosecond-level ($<$0.8 μs) packet scheduling and load-balancing under a variety of chaining configurations. We empirically show that PANIC can handle multi-tenant isolation and below line-rate offloads better than a state-of-the-art pipeline-based design. Our end-to-end experiments in a small scale testbed demonstrate that PANIC can achieve dynamic bandwidth allocation and prioritized packet scheduling at 100 Gbps. In total, the components of PANIC, which includes an 8 * 8 crossbar, only consume a total of 11.27% of the total logic area (LUTs) available on the Xilinx UltraScale Plus FPGA that we used. The Verilog code for our FPGA prototype is publicly available [1].

Offload	Config	Tput (Gbps)	Delay
Data Processing			
Compression (Lzrw1)	HW@300MHz	3.6	0.05-3.3μs
Cryptography (AES-256)	HW@300MHz	38.4	407ns
Cryptography (AES-256)	CPU@1.5GHz	0.154	–
Network Processing			
Authentication (SHA1)	HW@220MHz	113.0	0.47-10.8μs
Authentication (SHA1)	CPU@1.5GHz	0.192	–
Application Processing			
Inference (3-layer-NN)	HW@200MHz	120	66ns

Table 1: **A breakdown of the performance of different offloads when implemented in either hardware or software.**

2 Motivation

We discuss in detail the requirements that we envision programmable NICs in multi-tenant networks ought to meet. We then explain why existing NICs designs fail to meet them.

2.1 Requirements

1. Offload Variety: There are a large variety of network offloads, and different types of offloads have different needs. Not all offloads are best implemented on the same type of underlying engine. For example, a cryptography offload can provide much better performance if implemented with a hardware accelerator built from a custom IP core instead of an embedded processor core. To shed light on this, we experimented with a few different types of offloads using an Alpha Data ADM-PCIE-9V3 Programmable NIC [12] to evaluate the behavior of different hardware IP cores that could be used as on-NIC accelerators, and the Rocket Chip Generator [14] to perform cycle-accurate performance measurements of a RISC V CPU to understand the costs of running these offload with an on-NIC embedded processor. Our results in Table 1 indeed show that offloads for encryption/decryption and authentication are a poor fit for embedded CPU designs and should be implemented in hardware.

In contrast, an application-specific offload to walk a hash table that is resident in main memory is better suited for an embedded processor core because a hardware offload may not provide enough flexibility [51]. Thus, a programmable NIC should ideally provide support for both hardware and software offloads.

2. Dynamic Offload Chaining: In the case of hardware accelerators, it is important to be able to compose independent offload functionality into a chain/pipeline to avoid wasted area on redundant functionality. For example, using a programmable NIC to implement a secure remote memory access for a tenant may require the tenant to compose cryptography, congestion control, and RDMA offload engines.

[1] PANIC artifact: https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact

Further, as tenants come and go, and as a given application's traffic patterns change, the on-NIC offload chains will also need to be dynamically updated. This is because different network transfers benefit from different sets of offloads. Further, not every application packet needs every offload. For example, for a key-value store that serves requests from both within-DC and WAN-distributed clients, IPSec and/or compression could be offloaded, but only the packets sent over the WAN may need IPSec authentication and/or compression.

Thus, an ideal programmable NIC should not restrict the type of offloads that may be simultaneously used, and should instead support dynamic offload chaining, i.e., switching and scheduling packets as needed between independent offloads.

3. Dynamic Isolation: Today's data center servers colocate applications from different competing tenants [50, 39, 15, 61, 32]. Each tenant may have its own offload chains that may need to run on a programmable NIC, so it is necessary for a programmable NIC to provide performant low-level isolation mechanisms. For example, consider the case that two tenants A and B are running offload chains where packets are first uncompressed and then sent to an embedded CPU for further processing, and packet contents are such that the workload for tenant B runs at half the rate of that of tenant A. To support this, the NIC's mechanisms must ensure fair packet scheduling at the shared compression offload and that the slow chain does not cause head-of-line (HOL) blocking for the other chain. Further, if a third tenant C were to start, packet processing load across chains may shift. To handle this, the scheduling policy may need to be reprogrammed.

4. Support for offloads with variable and below line-rate performance: Some offloads may not run at line-rate. Of the compression, cryptography, authentication, and inference offloads that we ran on hardware, only inference was able to run at 100 Gbps (Table 1), and others ran well below line-rate. Also, offload performance is variable and sometimes workload-dependent, incurring significant delay for certain requests; see, for example, compression and authentication, whose performance depends on packet size.

These results also show the need for an approach to load-balancing that can accommodate offloads with variable performance. Slow offloads can be duplicated across multiple engines (e.g., 3 AES-256 engines) for line-rate operation.

5. High-Performance Interconnect: It is important for a programmable NIC to be able to provide high throughput for line-rate offloads. In the case where no offloads or only low-latency offloads are used, a programmable NIC should not incur any additional latency. Achieving high performance is complicated by bidirectional communication, multi-port NICs, and chaining. An offload that is used for TX and RX on a dual port NIC needs to operate at four times line-rate to prevent becoming a bottleneck. When offloads are chained, a single packet may traverse the on-NIC network multiple times. Effectively, the NIC must be able to emulate creating a line-rate connection between each hop in an offload chain.

NIC Design	Offload Chaining	Multi-Tenant Isolation	Variable Perf	High Perf	Offload Variety
Pipeline	✗	✗	✗	✓	✗
Manycore	✓	✗	✓	✗	✓
RMT	✗	✓	✗	✓	✗

Table 2: **Programmable NIC designs compared w.r.t. the requirements in Section 2.1.**

2.2 Limitations of Existing Designs

We argue below that programmable NIC designs today (Figure 1) lag behind these requirements (Table 2).

2.2.1 Pipeline Designs

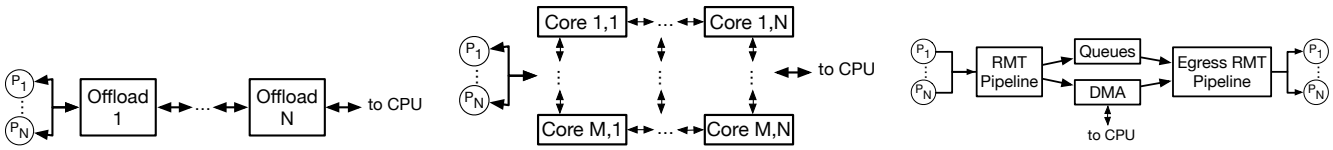
Figure 1a illustrates the pipelined programmable NIC design. In this design, the offloads are arranged in a linear sequence, *i.e.*, a pipeline. Effectively, each offload looks as though it is an independent device attached in the middle of the wire connecting the NIC to a TOR switch. Most existing NICs with on-board FPGAs located as a "bump-in-the-wire" use this design [52, 32, 31], and other NICs use this design for fixed function offloads for TCP checksums and IPSec [6, 38].

Chaining: Chaining offloads is difficult in pipelined designs because of their static offload topology; the offloads are arranged in a line. Although packets can be *recirculated* through the pipeline as needed, this wastes on-NIC bandwidth and hurts line-rate performance.

Variable Performance Offloads: A slow offload that does not run at line-rate can cause HOL blocking in the pipeline of offloads if the pipeline is stalled, and packet loss if the pipeline is not. This can be avoided with routing logic to bypass offloads, but this requires additional buffer memory at each offload: packet arrivals in Ethernet are bursty [41, 17], and it common for tens of packets to arrive back-to-back at line-rate. There would be significant packet loss if offloads that are not guaranteed to run at line-rate are not allocated buffer resources. For offloads where running at line-rate is workload or configuration dependent, the chip area allocated to per-offload buffers would be wasted under some traffic patterns.

Multi-tenant Isolation: In a pipeline, packets are forwarded through offloads that do not need to process the packet. Even if every offload runs at full line-rate, a high latency offload used by Tenant A but not by Tenant B will unnecessarily lead to increased latency for Tenant B. This can be avoided with routing logic to bypass offloads, but this also requires additional buffer memory at each offload to avoid pipeline stalls or packet drops. It is only possible to bypass an offload without stalling the pipeline if there is somewhere else to put the packets that it is currently processing.

Multi-tenant isolation is more problematic if not all offloads are guaranteed to run at line-rate. In this case, if tenant A has already consumed all of the packet buffers allocated for an offload, then tenant B will experience HOL-blocking and possibly packet loss. Although per-offload scheduling



(a) **A pipeline-of-offloads architecture** (b) **A tiled manycore NIC architecture** (c) **A NIC with RMT pipeline [42]**

Figure 1: **Illustrations of existing programmable NIC architectures.**

logic could be used to overcome this limitation, this has area overheads, and, as with per-offload packet buffers, this logic may be unutilized in some workloads.

Offload Variety: Pipeline-of-offloads designs are typically used for programmable NICs that only support hardware offloads. The limitations of pipeline designs are best avoided with low-latency offloads that run at line-rate. Because embedded CPU cores may not run at full line-rate and can incur high processing latency, this makes them a poor fit for pipeline designs. To overcome this limitation, the Azure SmartNIC [32] onloads computation from the programmable NIC to a core on the main CPU for certain tasks. This approach is costly, especially in cloud environments where servers are leased to customers on a per-core basis.

Some FPGA NICs implement all NIC functionality on an FPGA, including the Ethernet MAC and PCIe engines [12, 75, 76, 31]. Such NICs do not have many inherent limitations as a platform. With the right design, such NICs can meet all of our requirements, but no such design currently exists.

2.2.2 Manycore Designs

Figure 1b illustrates a manycore programmable NIC design [24, 53, 72, 73]. These designs implement network offloads by parallelizing flow processing across a large number of embedded processors that are arranged into a multi-hop on-chip tiled topology. Some manycore NICs additionally contain hardware engines for cryptography and compression [72, 58]. This supports chaining and a variety of different offloads, but performance and isolation are poor.

Performance: Manycore NICs use an embedded CPU core to orchestrate the processing of a packet across offloaded functions [34]. This is because the on-chip network cannot parse complex packet headers to determine the appropriate on-NIC addresses for the packet's destination. As a result of this design choice, manycore NICs have throughput and latency problems that prevent high-performance chaining. Further, manycore NICs even struggle to drive 100 Gbps and faster line-rates [32]. Because a single embedded processor is not enough to saturate line-rate, manycore NICs require packet load-balancing and buffering to scale performance.

Manycore NICs struggle to provide high-throughput chaining because manycore interconnects typically only provide enough throughput for a received packet to be sent to one embedded core before being sent via DMA to main memory. As applications become complex, state and caching limitations can require that different offloads be implemented as microservices distributed across cores instead of parallel monoliths. Current manycore NIC designs are not able to provide high performance for such a usecase.

Similarly, involving a CPU in a manycore NIC adds significant packet processing latency that otherwise could be avoided for packets that only need to be processed by a hardware accelerator. For example, Firestone *et al.* [32] report that processing a packet in one of the cores on a manycore NIC adds a latency of 10 μs or more.

Multi-Tenant Isolation: Because manycore NICs must buffer packets and load-balance them across parallel embedded cores [48], the extent to which tenants are isolated is determined by how buffer resources are managed, and how packets are load balanced. Unfortunately, existing manycore NIC designs do not provide explicit control over packet scheduling and buffering [48]. They use FIFO packet queuing and drop-tail buffer management; without any other isolation mechanisms, this can lead to HOL blocking, and drop-tail packet buffers can allow one tenant to unfairly consume buffers.

However, some level of isolation is possible in manycore NICs by (1) statically partitioning CPU resources across different tenants [48], which is inefficient, and (2) then using NIC-provided SDN mechanisms for steering tenants' flows to different cores. Additionally, some NICs such as the Broadcom Stingray allow running an OS to provide software-based isolation through a Linux operating system [22], but this can exacerbate the NICs' performance issues.

2.2.3 Reconfigurable Match+Action (P4) Designs

Figure 1c shows an RMT NIC design; these are built using an ASIC substrate with a programmable match+action (RMT) pipeline [42, 60]. In this model, incoming packets are first parsed by a programmable parser and then sent through a pipeline of M+A tables. Unfortunately, RMT NICs cannot support many interesting offloads (e.g., compression, encryption, or any offload that must wait on the completion of a DMA from main memory) because the actions that are possible at each stage of the pipeline are limited to relatively simple *atoms* that can execute within 1-2 clock cycles [67, 60, 21]. However, RMT NICs do not suffer from multi-tenant performance isolation problems because each offload runs at line-rate with extremely low latency.

3 PANIC Overview

PANIC is a new programmable NIC design that meets the aforementioned requirements (Section 2.1). The core idea be-

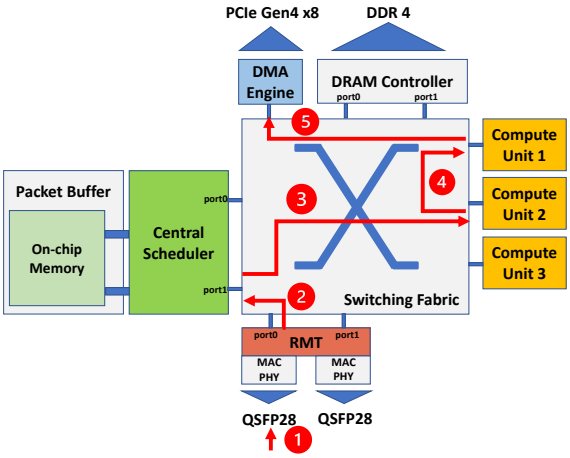


Figure 2: **PANIC Architecture**

hind the design of PANIC is that programmable NICs should be implemented as four logical components (Figure 2): 1) A programmable RMT pipeline, which provides programmable offload chaining on a per-packet basis; 2) A switching fabric, which interconnects all other components in PANIC and enables dynamic chaining at line-rate; 3) A central scheduler, which achieves high-performance packet scheduling, traffic prioritization and traffic isolation; 4) Compute units, each of them running a single offload. This system architecture is shown in Figure 2. We show that this design is suitable for both ASIC and FPGA implementations.

Operational overview: Figure 2 illustrates how PANIC operates when packets are received. In this example, there are three compute units 1, 2, and 3 running services A, B, and C respectively. When packets are received in PANIC in Step 1, they are first processed by the RMT pipeline. The RMT pipeline parses the packet headers and matches on them to identify the chain of offloads that the packet should be forwarded to, and then it generates a PANIC descriptor that contains this offload chain information. In this example, the offload chain that is found will first send the packet to service B and then to service A.

Next, the packet is injected into the switching fabric. If the packet does not need to be processed by any offloads, it will be forwarded straight to the DMA engine of the NIC, which is connected to the interconnect in the same manner as all of the compute units used to implement offloads. Otherwise, it is sent to the central packet scheduler (Step 2).

The scheduler then buffers the packet and orchestrates scheduling and load-balancing the request across its offload chain. When there is no load, packets are chained with a source route that takes them from offload to offload without stopping at the packet scheduler. In this example, the scheduler first buffers this packet until Unit 2 is idle. Then, in Step 3, it steers this packet to Unit 2, and, in Step 4, the packet is directly *pushed* to Unit 1. Finally, in Step 5, after Unit 1 finishes the computation of service A, the source route steers this packet to the DMA engine, which is responsible for transferring packets over the PCIe bus into main memory on the host.

When load is high (not shown), the loaded unit (say Unit 1) detours a packet that was pushed to it (by Unit 2) off to the buffer in the central scheduler. From there, the packet can be *pulled* later for processing by either Unit 1 when it has finished processing a packet or by another parallel unit running the same logic as Unit 1 entirely.

Transmitting packets is similar to receiving packets in reverse, except that the main CPU can associate offload chains with transmit queues beforehand so that the RMT pipeline does not need to process packets before they can be sent to offloads. After the main CPU enqueues packet descriptors, they will be read by the DMA engine, forwarded through an offload chain and managed by the central packet buffer as needed, and then forwarded to the appropriate Ethernet MAC.

PANIC makes it possible to meet all of our requirements:

1. Offload Variety: Each offload in PANIC is an independent tile attached to the high-performance interconnect, and the RMT pipeline builds the packet headers necessary to enable hardware offloads to process packet streams without any additional routing or packet handling logic. This allows for a large variety of different types of computation to be performed by the offload engine, including hardware IP cores, embedded processors, and even embedded FPGAs [74].

2. Dynamic Offload Chaining: Installing a new chain in the RMT pipeline for received packets involves programming lookup tables, and installing a new chain for a transmit queue can be done by issuing MMIO writes from the main CPU.

3. Policies for Dynamic Multi-Tenant Isolation: Performance isolation is provided by the central packet scheduler, which performs packet scheduling across the packets buffered for groups of parallel offloads that provide the same service. The scheduling algorithm determines both how chains competing for a service are isolated and how chains share packet buffers. Similar to prior work [68, 70], packet scheduling policies in PANIC are programmable. Further, PANIC improves upon prior work by also providing policy-aware packet dropping to enable cross-tenant memory isolation.

PANIC supports any scheduling algorithm that can be implemented by assigning an integer priority to a packet, and this includes a wide range of different policies, including strict priority, weighted fair queuing (WFQ), least slack time first (LSTF), and leaky bucket rate limiting [56, 68]. Although strict priorities lead to starvation, this is intended—if there is enough mission-critical traffic to consume all available resources, then it is acceptable for competing best-effort traffic to starve. If starvation is undesirable, it can be avoided by using WFQ and rate-limiting.

PANIC can support a hierarchical composition of different scheduling algorithms, *e.g.*, fair sharing across tenants with prioritization of flows for each tenant, although this comes with additional hardware costs. More complex scheduling algorithms are also possible in PANIC because priorities for

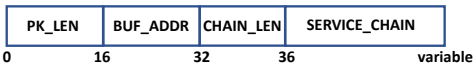


Figure 3: **PANIC Descriptor**

later services in a chain can be dynamically computed by an earlier chain stage. Similarly, each group of offloads that form a service can have its own custom scheduling algorithm, which is useful when different chains start with different offloads and then converge and share the same service.

4. Support for offloads with variable and below line-rate performance: The central packet scheduler supports offloads that have variable performance. Packets for slow offloads will be buffered at the central scheduler. As loads shift, packet buffers can be dynamically allocated to different offload groups. PANIC's hybrid push/pull load balancing scheme outlined in the example above load-balances packets across parallel offloads, ensuring precise load control, low tail latency, and minimal and efficient on-chip network use. Similar to the packet scheduler, the load balancer is also programmable.

5. High Performance: PANIC uses an on-chip network inspired by network routers to provide a high-performance interconnect between different offload tiles and the tiles for DMA and the Ethernet MACs. PANIC uses non-blocking high-bisection topologies like the crossbar making it possible to guarantee line-rate performance even if every offload in a chain sends/receives at line-rate over the on-chip network.

4 Design

This section discusses the design of the individual components of PANIC in more detail.

4.1 RMT Pipeline

The RMT pipeline in PANIC is used to provide programmable chaining and to look up scheduling metadata as part of providing programmable scheduling. The design of the RMT pipelines is borrowed from the design used in programmable switches [21, 67]. When a packet is received by the NIC, the RMT pipeline first parses incoming packets and then processes them with a sequence of match-action tables (MA tables). Each MA table matches specified packet header fields and then performs a sequence of actions to modify or forward the packet. Via these actions, the RMT pipeline 1) performs simple, line-rate packet processing (*e.g.*, IP checksum calculation) and 2) generates a PANIC descriptor for each packet that contains the appropriate chaining and scheduling metadata given the configuration that was programmed by the operator/user. Additionally, the RMT pipeline can maintain state on a per-traffic-class or per-flow basis if needed to support programmable scheduling or flow affinity.

Figure 3 shows the PANIC descriptor added by the RMT pipeline. It includes the packet length, the allocated buffer address, and the service chain for this packet, which is a list of services to send the packet to along with per-service metadata from the RMT. Because multiple compute units

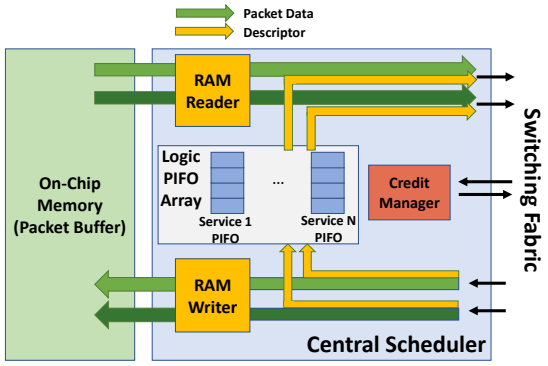


Figure 4: **Architecture of the multi-ported central scheduler**

may implement the same service (offload), this means that the RMT pipeline does not specify the exact unit a packet will be sent to in advance. This enables the scheduler to perform dynamic load balancing across multiple computation units implementing the same service in parallel.

In addition to specifying a list of services, the offload chain also contains metadata. For example, per-hop scheduling metadata like traffic class and priority allows a chain to have different priorities and weights across different services. Similarly, the descriptor may also contain service-specific metadata to allow the RMT pipeline to perform pre-processing to speed-up or simplify different compute units. Examples of this type of metadata include pointers to fields in a parsed packet and a pre-computed hash of an IP address.

The RMT pipeline directly connects to the switching fabric. To ensure low latency, the pipeline directly steers packets that are not processed by any service to the DMA engine.

4.2 High Performance Interconnect

To enable dynamic service chaining, PANIC use an on-chip interconnect network to switch packets, providing high-speed communication between the scheduler and on-NIC units.

Because it is necessary to forward packets between offloads at line-rate, it is important to build a high-performance network. PANIC utilizes a non-blocking, low latency, and high throughput crossbar interconnect network, which, for the scale of our design, still has a low area and power overhead. The crossbar can be configured to connect any input node to any output node with no intermediate stages, and each port runs at line-rate. As a result, every offload can simultaneously send and receive at line-rate, which enables line-rate dynamic chaining regardless of which offloads a chain uses.

Although economical in small configurations, crossbar interconnects unfortunately do not scale well with an increase in the number of cores. Most of these problems arise from the delay and area cost associated with long interior wires because the number of these wires increases significantly with the number of cores. Fortunately, with PANIC, we are able to choose between different interconnect topologies without having to change other parts of the design. If there is

a need to scale beyond the limits of a single crossbar, we can switch to a more scalable (but higher-latency) flattened butterfly topology [43].

4.3 Centralized Scheduler

The centralized scheduler buffers packets, schedules the order in which competing packets are processed by a service, and load-balances packets across the different compute units in a service. The scheduler architecture is shown in Figure 4. The scheduler uses a new hybrid scheduling algorithm to support low-latency chaining while avoiding load imbalance, and it uses a new hardware-based priority queue (*i.e.*, PIFO [68]) to schedule and drop packets according to a programmable inter-tenant isolation policy.

An overview of the operation of the central scheduler is as follows: When a packet and its descriptor arrive, the scheduler writes the packet data into high-speed on-chip memory and stores the packet descriptor into the appropriate logical PIFO queue given the next destination service of this packet. Each logical PIFO queue corresponds to a service and sorts buffered descriptors by rank, which enables the scheduler to drop packets according to the same policy as they are scheduled by dropping the lowest-rank packet currently enqueued for the service if needed. Then, whenever any of the parallel compute units for a service have available "credits" at the scheduler, the credit manager (Figure 4) chooses the compute unit with most credits, dequeues the head element of the corresponding logical PIFO queue, and sends the packet data and descriptor along with the packet data across the on-chip interconnect to the chosen unit.

4.3.1 Hybrid Push/Pull Scheduling and Load Balancing

When one service cannot achieve line-rate with a single unit, PANIC uses multiple parallel units to improve bandwidth. To support load-balancing across variable performance offloads, PANIC provides *load-aware* steering. Specifically, PANIC introduces a new hybrid pull/push scheduler and load balancer that overcomes the limitations of either push or pull scheduling to provide both precise request scheduling and high utilization.

Pull-based scheduling provides the most precise control over scheduling because decisions are delayed until each unit is able to perform work. However, pull-based scheduling can lead to utilization inefficiencies because each unit must wait for a pull to complete before it can start work on a new packet, and busy-polling can lead to increased interconnect load. In contrast, push-based scheduling can lead to load-imbalance and increased tail latencies when packets have variable processing times. In this scenario, it is not possible to know how much work is enqueued at each unit at the time that load-balancing decisions must be made.

The hybrid scheduler used in PANIC provides the best properties of both pull and push scheduling. In this scheduler, during periods of *high* load, the central scheduler uses *pull*-based load balancing to provide effective load balancing and packet scheduling. During *low* load, the scheduler *pushes* packets to all of the units in a service chain with low latency. To accomplish this, the scheduler uses credits to monitor the load at different units. Next, we describe the two modes of operation, pull and push, and the use of credits.

Credit Management: The credit manager tracks credits to measure load and dynamically switch between push-based and pull-based scheduling. The credit manager initially stores C credits for each compute unit. After sending a packet out, it decreases the credit number for that unit by one. When a compute unit is done processing a packet, it returns credit back to the credit manager.

The central scheduler operates in push mode as long as any of the parallel compute units in a service have credits available. If flow affinity is not needed, the scheduler steers packets to the unit which has the maximum number of pull credits to avoid load imbalance.

In contrast, if no unit has credit when a packet arrives, the scheduler buffers packets until credit is available. In this case, the central scheduler provides pull scheduling. Because the decision on which replica to use is made lazily, the number of packets queued at each unit will never exceed C.

By default, the number of initial credits C is set to two to avoid a stop-and-wait problem. However, it is possible to configure different credit numbers for each unit if needed. For example, ClickNP [47] uses a SHA1 engine that can process 64 packets in parallel, and PANIC can support this level of parallelism by giving 64 or more credits to the SHA1 engine.

Push-based Chaining: Push scheduling provides low-latency offload chaining. When a packet needs to traverse multiple offloads (*e.g.*, from A to B to C), the packet will be directly pushed to B when it finishes the computation in A rather than going back to the central scheduler. If B accepts the pushed packets, it will send a *cancel* message to the central scheduler to decrease its credit by one. In the case that there are multiple parallel units providing a service, the push destination is precalculated in the central scheduler. By pushing the packet directly to the next destination unit, PANIC reduces interconnect traversal latency and reduces on-chip network bandwidth demands. Furthermore, this reduces the load on the central scheduler as chain lengths grow.

Detour Routing: Push mode chaining may cause a packet to be pushed to a busy downstream unit that has no buffer space to accept packets. In this case, we use *detour* routing: when local buffer is occupied, the downstream unit redirects the packet back to the central scheduler, where it is buffered until it can be scheduled to another idle unit.

4.3.2 Packet Scheduling

To achieve priority scheduling and performance isolation, every packet stored in the packet buffer has its descriptor enqueued in a PIFO [68]. PANIC uses this PIFO-based priority

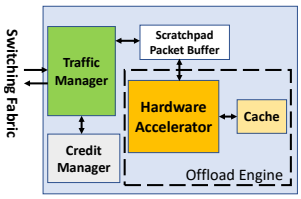


Figure 5: **Accelerator-Based Compute Unit Design**

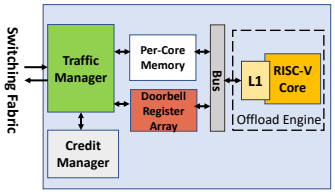


Figure 6: **Core-Based Compute Unit Design**

scheduler to provide both performance and buffer isolation across tenants.

Isolation Policy and Rank Computation: When a packet arrives, the central scheduler uses stateful atoms (ALUs) [67] to take metadata about the packet, look it up in the RMT pipeline, and compute an integer priority that is used to enqueue a descriptor for the packet into a PIFO block. This enables PANIC to provide multi-tenant isolation, ensuring traffic from high-priority tenants has low latency. Additionally, if multiple PIFO blocks are used inside the scheduler, it is possible to support hierarchical policies. Because the on-chip network ports are bidirectional, there is enough network throughput to forward incoming packets back out regardless of which logical queue the packets use.

Prioritized Dropping: PANIC's PIFO scheduler performs *prioritized packet dropping*. Specifically, PANIC ensures that when the NIC is overloaded, the lowest priority packets will be dropped. To achieve prioritized dropping, PANIC reuses the priority-sorted descriptor queue already used for scheduling in the PIFO. When the free space in the packet buffer for a logical PIFO is smaller than a threshold, the scheduler will remove the least-priority descriptor from the logical PIFO and drop this packet.

4.3.3 Performance Provisioning:

It is important to ensure that the central scheduler does not become a performance bottleneck and can forward packets across chains at full line-rate. To ensure that the scheduler has sufficient throughput, PANIC uses multiple ports to attach the scheduler to the on-chip network. Because the switching fabric is designed to forward between arbitrary ports at line-rate, increasing the number of ports used by the scheduler is sufficient to scale the network performance of the scheduler.

The speed of the PIFO block used to schedule packets can also become a performance bottleneck. The PIFO block that we use can schedule one packet per cycle, *e.g.*, 1000Mpps when operating at a 1GHz frequency when implemented in an ASIC design. Although this is sufficient to schedule packets in both transmit and receive directions in our current design, in the case that this number is greater than the performance of a single PIFO block, multiple parallel PIFO blocks need to be used to scale up performance.

Provisioning for Compute Unit Performance: The design of the on-chip network and the scheduler can also ensure that offloads may be fully utilized despite complications from chaining. Specifically, when an offload O1 in a chain (Chain A=O1–O2–O3) runs at a slower rate than the rest of the offloads (O2–O3), it will become a bottleneck and cause O2–O3 to be not fully utilized. However, this does not lead to resource stranding. A second chain B that does not use O1 can still use O2–O3 and benefit from the remaining capacity of these offloads. The scheduler can ensure that the contending chains fairly share capacity.

4.4 Compute Unit

To support offload variety, PANIC utilizes compute units to attach offloads to the switching fabric. These compute units are self-contained, meaning that hardware offloads can be designed without needing to understand the packet switching fabric and without having to issue pull requests to the hybrid scheduler. The interfacing with the switching fabric is handled by the traffic manager (Figures 5 and 6). This both reduces offload complexity by avoiding duplicating packet processing functionality and reduces packet processing latency by avoiding incurring the overheads of processing a packet with a CPU.

An offload engine in PANIC can either be a hardware accelerator or a core, and Figure 5 and Figure 6 presents the design of an ASIC accelerator-based and CPU-based compute unit in PANIC, respectively. In both of these designs, the offload functionality is encapsulated as an offload engine. Both perform packet processing by reading a packet once it arrives from the network and has been placed in a local scratchpad buffer. The traffic manager is responsible for communicating with the switching fabric. This component includes logic for sending and receiving packets as well as logic for updating PANIC descriptors as needed for push-based chaining. The compute unit's local credit manager interfaces with the central scheduler and is responsible for returning credits (when a packet's processing is done) and sending cancel messages that decrement credits (when accepting a pushed packet).

The primary difference between an accelerator-based design and CPU-based design is that there is additional logic in the CPU-based design that is used to interface with the memory subsystem of the embedded CPU core. As Figure 6 shows, the compute unit utilizes memory-mapped I/O (MMIO) to connect an embedded CPU core as follows: 1) The traffic manager (TM) writes network data directly to a pinned region of the per-core memory. 2) Then the TM writes to an input doorbell register to notify the core that data is ready. 3) After the core finishes processing, it writes data back to the pinned memory region if needed and then writes to an output doorbell register that is used to notify the TM of a new outgoing packet. To make sure the packet data is written back to memory, the core needs to flush the cache lines for the pinned memory region. 4) The TM collects the output data and sends it back to the switching fabric.

5 ASIC Analysis

We expect an eventual implementation of PANIC to use an application-specific integrated circuit (ASIC), although we have also prototyped PANIC in the context of an FPGA platform for expediency. This is because relative to an FPGA, an ASIC provides higher performance, consumes less power and area, and is cheaper when produced in large volumes [44]. While an ASIC implementation is beyond the scope of this work, in this section we briefly discuss the feasibility of implementing different components of PANIC in an ASIC.

RMT: The implementation of an RMT pipeline in ASIC has already been proven feasible [21]. The Barefoot Tofino chip [16] is a concrete realization of the RMT architecture.

PIFO: PANIC uses a hardware priority queue to provide programmable scheduling. Our current design was borrowed from the ASIC-based flow scheduler design of the PIFO paper [68]. While this design is conceptually simple and easy to implement because it maintains a priority queue as a sorted array, it is less scalable relative to other priority queue designs, e.g., PIEO [64], which uses two levels of memory or pHeap [20], which uses a pipelined heap. For better scalability, we can replace our current design with such scalable hardware designs of priority queues at the expense of greater design and verification effort.

Interconnect: One of the biggest potential scalability limitations of a PANIC implementation is the on-chip switching fabric. While crossbar interconnects are conceptually simple, the sheer number of wires in a crossbar might become a physical design and routing bottleneck, causing both an increase in area as well as an inability to meet timing beyond a certain scale. Fortunately, prior work has already demonstrated that it is feasible to build crossbars on an ASIC that are larger than are needed in PANIC. Specifically, Chole *et al.* built a 32 * 32 crossbar with a bit width of 640 bits [28, Appendix C] at a 1 GHz clock rate. As another data point, the Swizzle-Switch supports a 64 * 64 crossbar with a bit width of 128 bits using specialized circuit design techniques at 559 MHz in a relatively old 45 nm technology node [63]. For comparison, to provide 32 compute units each a 128 Gbps connection to the switching interconnect, PANIC only needs a 32 * 32 crossbar with a bit width of 128 bits and 256 bits at 1 GHz and 500 MHz clock frequencies, respectively.

At the same time, we anticipate a crossbar becoming no longer viable at some point as the number of offloads continues to increase. At this point, we anticipate switching to other more scalable topologies such as a flattened butterfly at the cost of increased latency and reduced bisection bandwidth.

Compute Units: The PANIC offload engine can be a hardware accelerator or a CPU core. There are several ASIC-based RISC-V implementations which can used as a CPU core for the offload engine [25, 26, 8]. Several functions important to networking, such as compression, encryption, and checksums are available as hardware accelerators, which can be reused for PANIC. Our own AES and SHA implementations (Section 6) are based on open-source hardware accelerator blocks [1, 7].

6 FPGA Prototype

We implement an FPGA prototype of PANIC in ∼6K lines of Verilog code, including a single-stage RMT pipeline, the central scheduler, the crossbar, the packet buffer, and compute units. Also, we built a NIC driver, DMA Engine, Ethernet MAC, and physical layer (PHY) using Corundum [33]. Although the PANIC architecture supports both the sending path and receiving path, in our current implementation, we mainly focus on the receive path.

RMT pipeline: We implemented a single-stage RMT pipeline in our FPGA prototype. We configure the RMT pipeline in our prototype by programming the FPGA. The RMT pipeline maintains a flow table, in which each flow is assigned an offload chain and scheduling metadata. In the match stage, the RMT module matches the flow table with the IP address fields and port fields in the packet header. In the action stage, the RMT module calculates scheduling metadata and generates the PANIC descriptor (Figure 3). The frequency of the RMT pipeline is 250 MHz.

FPGA-based Crossbar: We have implemented an 8 * 8 fully connected crossbar in our FPGA prototype. The frequency for this crossbar is 250 MHz, and the data width is 512 bits. This leads to a per-port throughput of 128 Gbps.

Central Scheduler and Packet Buffer: The architecture of the scheduler is shown in Figure 4. The scheduler is connected with two crossbar ports to ensure a sufficiently high throughput connection to the on-chip network. In our implementation, the PIFO block runs at 125 MHz frequency with a queue size of 256 packets; all other components in the scheduler run at 250 MHz, with a 512 bit data width, and we add a cross-domain clocking module between other components and the PIFO. We use lower frequency for the PIFO because it suffers from a scalability issue when implemented on the FPGA (we explain this further in Section 7.5). The packet buffer is implemented with dual-channel high-speed BRAM, where each BRAM channel supports concurrent reads and writes. The packet buffer size in our implementation is 256 KB with a 512 bit data width and a 250 MHz frequency. For ease of implementation, our current prototype uses a separate physical PIFO for each logical PIFO at the cost of increasing the relative resource consumption of PIFOs.

Compute Units: As Figure 5 shows, our implementation of a compute unit includes a traffic manager, a credit manager, and a scratchpad packet buffer. We choose the AXI4-Stream interface [2] as the common interface between the offload engine and scratchpad buffer. We have included two types of accelerator-based offload engines in our FPGA prototype. One is the AES-256-CTR encryption engine [1], and the other is the SHA-3-512 hash engine [7].

We have also implemented a RISC-V core engine based on the open-source CPU core generator [9]. Figure 6 shows how the RISC-V core is connected to PANIC's traffic manager, credit manager, and per-core memory. The RV32I RISC-V core we use has a five-stage pipeline with a single level of cache. The data cache and instruction cache are 2 KB each, and the local memory size is 32 KB. The frequency for this CPU is 250 MHz, and the per-core memory data width is 512 bits.

7 Evaluation

This section evaluates our FPGA prototype to show that it meets the design requirements put forth in Section 2.1. In Section 7.2, we use microbenchmarks to show that PANIC achieves high throughput and low latency under different offload chaining models. In Section 7.3, we compare PANIC's performance with a pipeline-of-offloads NIC. Section 7.4 measures the I/O performance of a RISC-V core in PANIC, and Section 7.5 measures the hardware resource usage of our FPGA prototype. Finally, in Section 7.6, we implement different offload engines, and test PANIC end-to-end; these results demonstrate that PANIC can isolate and prioritize traffic efficiently under multi-tenant settings.

7.1 Testbed and methodology

For our microbenchmarks, we implemented our FPGA prototype in the ADM-PCIE-9V3 network accelerator [12], which contains a Xilinx Virtex UltraScale Plus VU3P-2 FPGA. For this evaluation, we also implemented a delay unit, a packet generator, and a packet capture agent on the FPGA. The delay unit emulates various compute units by delaying packets in a programmable fashion, which allows us to flexibly control per-packet service time and chaining models. This enables us to run microbenchmarks that systematically study PANIC's performance limits. The packet generator generates traffic of various packet sizes at different rates. The packet capture agent receives packets and calculates different flows' throughput and latency. We calculate packet processing latency by embedding a send timestamp in every generated packet.

For our end-to-end experiments, we evaluate PANIC in a small testbed of 2 Dell PowerEdge R640 servers. One server is equipped with a Mellanox Connectx-5 NIC, and the other server is equipped with the ADM-PCIE-9V3 network accelerator carrying our PANIC prototype. The Mellanox NIC and ADM-PCIE-9V3 card are directly connected. We program the VU3P-2 FPGA on the network accelerator using our Verilog implementation of PANIC. We use DPDK to send customized network packets and use PANIC to receive packets and run offloads. Because of a performance bottleneck in the kernel-based FPGA NIC driver [33], we use the packet capture agent on the FPGA to report PANIC's receive throughput instead of capturing packets on the host machine.

7.2 PANIC System Microbenchmarks

We microbenchmark PANIC's performance using the different chaining models shown in Figure 7. Our results demonstrate that PANIC can both achieve high throughput and low latency for various common offload chaining models.

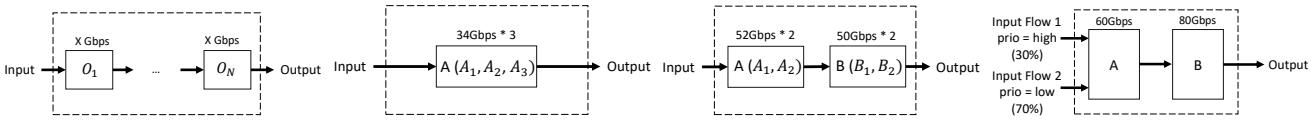
Model 1 ("Pipelined Chain"): In model 1 (Figure 7a), we attach N delay units in sequence. Each unit emulates a different service, and all of them process packets at X Gbps with fixed delay. We then configure a service chain that sends packets through all N units in numerical order.

First, we measure the throughput and latency overhead of PANIC when N = 3 and X = 100. Figure 8a shows the overall throughput under different packet sizes. With MTU-sized packets, PANIC can schedule packets at full line-rate. When the initial number of credits in PANIC is small, we see a nonlinear performance downgrade with small packets. This is because throughput for small packets is bounded by the scheduling round trip time in PANIC, which is 14 clock cycles. If we increase the initial credit number for each unit, we see a performance increase for small packets. When the credit number is greater than 8, the small packet performance is no longer bounded by the scheduling round trip, instead, it is bounded by the small packet performance of our delay unit. These results also demonstrate the benefits of PANIC's flexibility in per-unit credit allocation. Setting different credit numbers for each unit can improve performance.

Next, Figure 8b shows the latency of different packet sizes in the same experiment. In this pipelined chain, packets can be scheduled through three units within 0.5 microseconds. This low latency performance also arises from PANIC's push scheduling which helps PANIC avoid extra packet traversals between units and the scheduler.

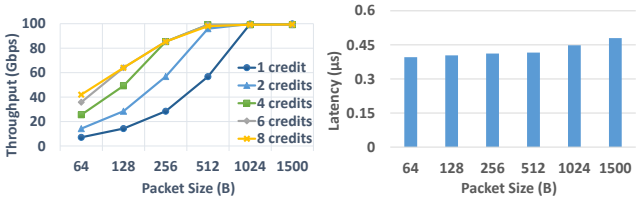
Next, Figure 9 shows PANIC's throughput as a function of the chain length when push scheduling is disabled. Without push scheduling, the packet needs to go back to the scheduler at every hop, and the total traffic that goes into the scheduler is the ingress traffic from the RMT pipeline plus the detoured traffic from units, which is: $T_{total} = T_{RMT} + T_{detour} = T_{RMT} + (N-1) * X = N * X$. As Figure 9 shows, when T_{total} exceeds the dual-ported scheduler's maximum bandwidth (250 MHz * 512 bits * 2 ports = 256 Gbps), the chaining throughput downgrades. For example, when N = 3, X = 70, $T_{total} = 210\ Gbps < 256\ Gbps$, thus PANIC can schedule this chain at full speed even when push scheduling is disabled. When N = 3, X = 90, $T_{total} = 270\ Gbps > 256\ Gbps$, the chaining throughput downgrades since the scheduler bandwidth becomes the bottleneck.

Model 2 ("Parallelized Chain"): In model 2 (Figure 7b), we attach three delay units running in parallel. These three units run the same service, and each unit has an average 34 Gbps throughput but *variable latency*. PANIC load-balances packets across these units. Figure 8c shows the throughput under different packet sizes and service time variance (the service time follows uniform distribution). We see that even when

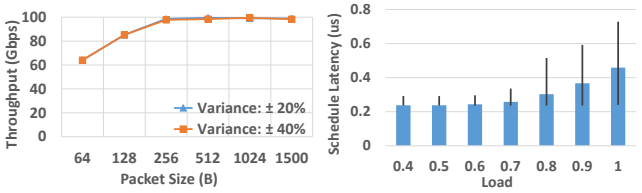


(a) **Chaining Model 1** (b) **Chaining Model 2** (c) **Chaining Model 3** (d) **Chaining Model 4**

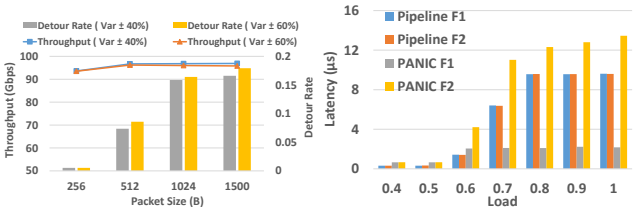
Figure 7: **The different chaining models used in experiments.**



(a) **Model 1 throughput** (b) **Model 1 latency**



(c) **Model 2 throughput** (d) **Model 2 latency**



(e) **Model 3 detour rate and throughput.** (f) **Different flow's latency in the pipeline design and PANIC**

Figure 8: **PANIC performance under different chaining models**

the processing latency variance increases from 20% to 40%, PANIC can still efficiently load-balance packets between the parallel units without impacting throughput. In this experiment, small packet performance is better than model 1 because model 2 has multiple units running in parallel. Overall throughput is no longer bounded by the delay of a single unit.

Figure 8d shows PANIC scheduling latency under different loads with MTU sized packets and 40% service time variance. The error bars in this figure represent 5%-ile and 99%-ile latency. Scheduling latency reveals how long the incoming packets wait before being processed by an idle unit; we calculate it by subtracting the unit processing time from the total latency. When the NIC load is much smaller than 1, scheduling tail latency grows slowly, and is under 0.4 μs. When the load approaches 1, queueing occurs in the packet buffer, which causes tail latency to grow, but it still stays < 0.8 μs. This shows that our credit-based scheme keeps latency low even at high load, and most latency is due to queueing.

Model 3 ("Hybrid Chain"): Model 3 (Figure 7c) is a hybrid chaining model where packets are not only load-balanced between parallel units but also go through multiple services.

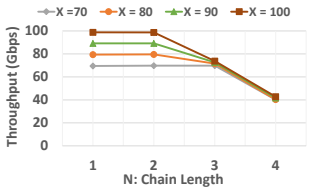


Figure 9: **Model 1 throughput when push is disabled.**

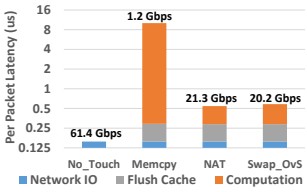


Figure 10: **Performance of a single RISC-V core with MTU-sized packets.**

	Throughput	Total
Flow 1 (Pipeline Design)	18.7 Gbps	60.6 Gbps
Flow 2 (Pipeline Design)	41.9 Gbps	
Flow 1 (PANIC)	30.7 Gbps	59.8 Gbps
Flow 2 (PANIC)	29.1 Gbps	

Table 3: **Throughput of the pipeline design and PANIC.**

Packets need to be processed first by service A and then by service B, and both services have multiple parallel compute units. Each compute unit for service A has an average throughput of 52 Gbps, while each compute unit for service B has an average throughput of 50 Gbps. Compute units for both service A and B have variable latency.

Figure 8e shows the throughput and detour rate in this hybrid model. When the packet size is bigger than 256 bytes, the detour rate is high. This is because the downstream B units have lower throughput than the upstream A units. As a result, the B units are always busy because they are the throughput bottleneck in this system. Busy units are likely to have no space to accept pushed packets: if A unit tries to push packets to a busy downstream B unit, then the B units will more often than not detour the pushed packets back to the central scheduler.

Figure 8e also shows that detour routing does not degrade throughput. This is because the maximum bandwidth of our dual-ported scheduler is 256 Gbps, and in this hybrid model, the ingress traffic from the RMT pipeline will take up 100 Gbps bandwidth in the scheduler, thus there is more than 100 Gbps bandwidth left for the detoured traffic.

However, detour routing can increase packet latency. In order to mitigate this, the central scheduler increases the priority for each detoured packet, to help them get rescheduled first. Thus, the latency incurred by detoured packets is the RTT between the compute unit and the scheduler, which is < 0.5 μs.

7.3 Comparison with the Pipeline Design

To demonstrate that PANIC handles multi-tenant isolation and below line-rate offloads better than state-of-the-art, we build and compared against the pipeline-of-offloads NIC as our baseline. We choose model 4 (Figure 7c) as the offload chain for this comparison. The difference between model 4 and model 1 is that the delay unit emulates a below-line-rate offload in model 4. We assume two flows are competing: Flow 1 has a higher priority, and takes up 30% of the total traffic. Flow 2 has a lower priority, and takes up 70% of the total traffic.

We implemented a pipeline-of-offloads NIC in the ADM-PCIE-9V3 network accelerator. In this NIC, all incoming packets are first buffered in a FIFO (First-In-First-Out) queue before entering unit A. Unit A and unit B are directly connected using the AXI4-stream interface [2]. We configured the pipeline-of-offloads NIC and PANIC to have the same buffer size (64 KB), same frequency (250 MHz), and same bit-width (512 bits).

Figure 8f presents a comparison of the latency of both the pipeline design and PANIC in this experiment. When the NIC load is low, Unit A is not the bottleneck, and both NICs have low latency. The pipeline design has slightly better latency since units are directly connected in it, while scheduling packets in PANIC has some overhead. When load increases, Unit A becomes the bottleneck, and both NICs start to buffer and drop packets. With high load, Flows 1 and 2 have the same latency in the pipeline design, since packets are scheduled in First-Come-First-Served order and can experience HOL blocking. In PANIC, the high priority packets have fixed low latency due to the central scheduler sorting buffered packet descriptors and serving high priority packets first.

We compare the throughput between the pipeline design and PANIC in Table 3. The total throughput is bounded by Unit A (60 Gbps). In the pipeline design, the low priority flow 2 has a higher throughput than flow 1, because the high-volume flow 2 steals on-chip bandwidth by taking up most of the on-chip buffer. PANIC preferably allocates buffer to high priority packets and always drops the lowest priority ones. Thus flow 1 can always achieve full throughput in PANIC.

Overall, PANIC achieves good isolation: 1) PANIC achieves comparable throughput and latency with the pipeline design when there is no HOL blocking. 2) When HOL blocking occurs, PANIC ensures that the high priority flows have a fixed low latency. 3) PANIC allocates bandwidth according to a flow's priority.

7.4 RISC-V Core Performance

To investigate the I/O overhead of using an embedded NIC core to send/receive network packets from PANIC, we performed experiments with a single RISC-V CPU core as the only offload engine in a chain. We measure the system throughput and per-packet latency using four example C programs:

No-Touch: After receiving the packet from PANIC, this program will send the packet back to PANIC immediately. This program does not make any changes to the packet data.

Memcpy: This program will copy the received packet to another memory address and then send the copied packet back to PANIC.

NAT: The Network Address Translation (NAT) program uses the embedded CPU core to lookup a <Translated IP, Port>pair for a given 5-tuple, and then replace the IP address and port header fields using the lookup results. The lookup table is stored in the local memory inside the offload engine. The RMT pipeline will pre-calculate the hash value for each packet, and the hash value is stored as per-service metadata in the PANIC descriptor. Thus the CPU core can directly read the pre-calculated hash value from the descriptor.

Swap OvS: This program swaps the Ethernet and IP source and destination addresses.

Figure 10 shows the RISC-V core throughput and per-packet latency with MTU sized packets. We breakdown the latency number into three different parts: 1) Network I/O: the time that is spent on pulling/writing the input/output doorbell register, 2) Flush Cache: the time spent on flushing the L1 cache, 3) Computation: the time spent on computation, including the data exchange time between the L1 cache and the per-core memory. The results of this experiment show that the overhead of the NIC to CPU core interface is low, and, for those low-throughput applications, the I/O time introduced by PANIC is negligible.

For example, the throughput of the No-Touch program is 61.4 Gbps, and all the time is spent in network I/O. The throughput of the NAT and Swap OvS programs is 21.3 Gbps and 20.2 Gbps, respectively. ∼20% of the time is spent in flushing the cache, ∼27% in network I/O, and ∼50% in computation. Cache flushing is costly in our current prototype: to synchronize the data between the cache and memory, the whole L1 cache is flushed before processing the next packet. If needed, this performance could be improved by modifying the CPU to support an instruction that only flushes the cache lines for the pinned memory region used by the packet.

The throughput of Memcpy is only 1.2 Gbps, and 97% of the time is spent in computation. This is due to the limitations of the performance of the FPGA based RISC-V core. With a faster core and a higher clock frequency, this performance can be improved.

7.5 Hardware Resource Usage

Our UltraScale VU3P-2 FPGA has 3 MB BRAM, and 394k LUTs in total. Table 4 shows different components' resource usage under different settings. In our end-to-end experiments (Section 7.6), the crossbar has 8 ports, total queue size in the PIFO array is 256 packets, and packet buffer size is 256 KB. Under this setting, we find that PANIC's design will only

Module	Setting	LUTs(%)	BRAM(%)
Crossbar	8 ports	5.5	0.00
	16 ports	13.64	0.00
Scheduler (PIFO)	PIFO = 256	5.18 (4.9)	0.07 (0.01)
	PIFO = 512	9.95 (9.42)	0.07 (0.01)
Packet Buffer	256 KB	0.16	8.94
Simple RMT	/	0.27	0.00

Table 4: **FPGA resource usage for different components.**

cost 11.27% logic area (LUTs) in our middle-end FPGA. Total BRAM usage is 8.94% due to the limited BRAM in our FPGA.

The crossbar and PIFO occupy most of the on-chip logic resources in PANIC. When the crossbar uses 8 ports, it costs around 5.5% logic area, and for 16 ports, the logic area cost is 13.64%. When the total PIFO size is 256, it will cost 4.9% logic area, and when the size is 512, it will cost 9.42%. PIFO suffers from high logic area cost because its hardware design does not access BRAM at all; it only uses the logic unit to compare and shift elements. This design causes PIFO to be less scalable in the FPGA since it cannot benefit from the FPGA's memory hierarchy to efficiently distribute storage and processing across SRAM and LUTs. Recent advancements [64] can be used to address this (Section 5). Overall, we find that PANIC can easily fit on any middle-end FPGA without utilization or timing issues.

7.6 End-to-End Performance

In this section, we measure PANIC's end-to-end performance in our cluster. Because of the performance bottleneck of the kernel-based FPGA NIC driver, we use hardware counters to measure PANIC's receiving throughput. We implement two FPGA-based offload engines in PANIC: a SHA-3-512 engine and an AES-256 engine. Our end-to-end experiment demonstrates that: 1) PANIC can schedule network traffic at full line-rate, 2) PANIC can precisely prioritize traffic when different flows are competing for computation resources at the offloads, and 3) PANIC can support different isolation policies, including strict priority and weighted fair queueing.

The AES-256-CTR encryption engine [29] encrypts input plain text into ciphered text or decrypts ciphered text to yield plain text. The fully pipelined AES-256 engine can accept 128-bit input per cycle, and it can run at 250 MHz frequency with 32 Gbps throughput. The SHA-3-512 engine [19] performs SHA-3, a newest cryptographic hash which uses permutation as a building block [18]. The FPGA-based SHA-3-512 engine that we use runs at 150 MHz with 6 Gbps throughput.

Since the throughput of a single SHA engine is low, we put 4 SHA engines into a single hash unit, and set the initial credit number for the hash unit to 4. Thus, the hash unit can use 4 SHA engines to process these packets in parallel. We connect two decryption units and two hash units with PANIC. Thus, the bandwidth of hash computation is (6 * 4) * 2 = 24 * 2 = 48 Gbps, and that for decryption is 32 * 2 = 64 Gbps.

In our experiment, we assume there are two types of traffic competing for the computation resource in PANIC. One is high-volume multimedia traffic, which uses AES offload to decrypt video streams. Another is low-volume IPSec traffic, which first uses SHA to ensure the integrity of the data and then uses AES to decrypt IP payload. The IPSec traffic has higher priority than video stream traffic, and each of these traffic streams contains multiple flows. Also, we add background traffic that does not need to be processed by any compute unit. The offload chains are shown in Figure 12.

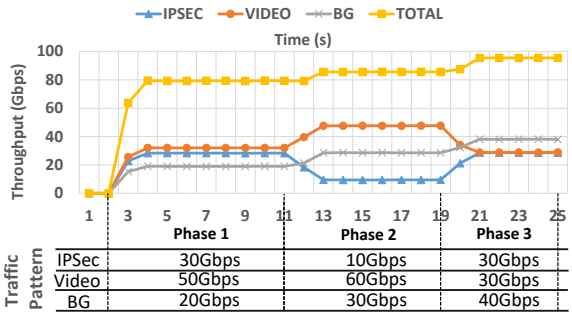
Traffic	IPSec	Video	Background
Drop Rate	0%	33.1%	0%

Table 5: **Packet dropping rate in phase 1 in Figure 11a.**

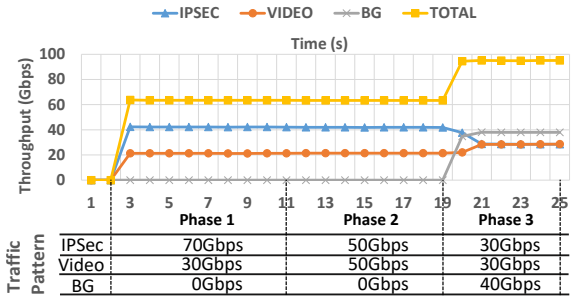
In the first experiment, we use the strict priority policy, which means all the IPSec packets have higher priority than the video packets. Figure 11a shows different traffic's receiving throughput under different traffic patterns. In phase 1, the sending throughput is 30 Gbps for IPSec and is 50 Gbps for video. We can see the receiving throughput for IPSec is 30 Gbps, which is the same as the sending throughput. The receiving throughput for the video stream is only 34 Gbps. This is because IPSec and video stream share the AES offload. However, the available peak bandwidth for the AES offload is only 64 Gbps. Thus, PANIC will first satisfy the high priority IPSec traffic requirement, which only leaves 34 Gbps (64 - 30) of bandwidth for the video stream. Table 5 shows the dropping rate under phase 1. Due to prioritized packet dropping, PANIC only drops low priority video packets. Overall, when a below-line-rate offload becomes the bottleneck, PANIC always first satisfies high priority traffic's bandwidth demands.

In phase 2, the DPDK sender switches to the next traffic pattern, in which the IPSec traffic sending rate drops to 10 Gbps, and video traffic sending rate grows to 50 Gbps. Since the IPSec sending rate drops, the video stream can get more bandwidth, but it will still lose some bandwidth and experience packet drops because of the AES bottleneck. In phase 3, the DPDK sender switches to the last pattern, in which the AES offload is no longer the bottleneck; no packet drops occur, and the total throughput can reach 100 Gbps. Another noteworthy aspect in Figure 11a is that no matter what computation happens, background traffic performance is unaffected.

In the second experiment, we use a weighted fair queueing (WFQ) scheduling policy where the AES offload's capacity is divided across IPSec traffic and video traffic in 2 : 1 ratio. Figure 11b shows the throughput of the different traffic types under the WFQ policy for different traffic patterns. In phase 1, the sending throughput for IPSec is 70 Gbps, and for the video stream is 30 Gbps. We can see the receiving throughput for IPSec is exactly twice of the video stream, and the total throughput is 64 Gbps. If the IPSec sending rate drops to 50 Gbps (phase2), the receiving throughput remains unchanged. This result proves PANIC can shape the traffic



(a) **Strict Priority Policy**



(b) **Weighted Fair Queueing Policy**

Figure 11: **Receiving throughput with different traffic patterns. Figure a uses strict priority policy: all the IPSec packets have higher priority than the video packets. Figure b uses WFQ policy: the offload capacity is divided across IPSec traffic and video traffic in the ratio 2:1. The table in Figure a and Figure b shows how the sending traffic pattern changes with time.**

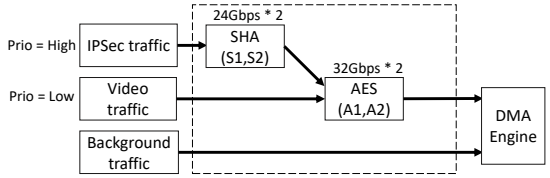


Figure 12: **Offload chains; end-to-end experiment**

precisely, regardless of the sending rate.

In phases 1 and 2, the scheduler switches to pull-based scheduling since the AES offload is always congested. As a result, the egress packet of the SHA offload goes directly back to the scheduler instead of the congested AES offload. The scheduler then shapes the video traffic and the detoured IPSec traffic into a desired rate using WFQ.

In phase 3, the AES offload is no longer the bottleneck. Thus, the central scheduler operates in push mode: the egress packet of the SHA offload can bypass the scheduler and be directly pushed to the AES offload. As shown in Figure 11b, both IPSec and video's receiving throughput can reach the sending rate, which is 30 Gbps. Overall, this shows that PANIC can shape the traffic precisely with the WFQ policy.

8 Related Work

Several projects introduce new offloads that utilize programmable NICs and new frameworks for deploying these offloads [13, 48, 59, 42, 32, 37, 49, 65, 46, 62, 47, 40, 30, 36, 70, 69, 35, 55, 45]. PANIC is orthogonal to these projects.

The Pensando DSC-100 NIC [58] is similar to PANIC in that it has an RMT pipeline and supports both hardware and software offloads. However, the DSC-100 requires cores to achieve offload chaining instead of a hardware scheduler.

The Fungible Data Processing Unit (DPU) is a NIC design that was recently announced in August 2020 [3]. Based on publicly available documents [4, 5], it has a hardware architecture that shares a few similarities with PANIC (e.g., processing cores, accelerators, a hardware work scheduler, and a customized on-chip network). A head-to-head comparison of PANIC to the Fungible DPU would be an interesting avenue for future work once the DPU is generally available.

PANIC is also similar to FairNIC [34], which improves fairness between competing applications running on a commodity manycore NIC. However, PANIC provides features not possible in FairNIC like chaining without involving a CPU. Further, FairNIC helps motivate the need for PANIC detailing the non-trivial costs of isolation on manycore NICs. Adopting PANIC's scheduler and non-blocking crossbar interconnect can solve these fundamental problems with manycore NICs.

9 Conclusions

Programmable NICs are an enticing option for bridging the widening gap between network speeds and CPU performance in multi-tenant datacenters. But, existing designs fall short of supporting the rich and high-performance offload needs of co-resident applications. To address this need, we presented the design, implementation, and evaluation of PANIC, a new programmable NIC. PANIC synthesizes a variety of high-performance hardware blocks and data structures within a simple architecture, and couples them with novel scheduling and load balancing algorithms. Our analysis shows that PANIC is amenable to an ASIC design. We also built a 100G PANIC prototype on an FPGA, and conducted detailed experiments that show that PANIC can isolate tenants effectively, ensure high throughput and low latency, and support flexible and dynamic chaining.

Acknowledgements: We thank our shepherd, Costin Raiciu, and the anonymous OSDI reviewers for their feedback that significantly improved the paper. We thank Suvinay Subramanian and Tushar Krishna for discussions on crossbar designs and Tao Wang for his assistance with the artifact evaluation. Brent E. Stephens and Kiran Patel were funded by a Google Faculty Research Award and NSF Award CNS-1942686. Aditya Akella and Jiaxin Lin were funded by NSF Awards CNS-1717039 and CNS-1838733 and a gift from Google.

References

[1] AES Hardware Accelerator. https://opencores.org/projects/tiny_aes.

[2] Axi reference guide. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[3] Fungible DPU: A New Class of Microprocessor Powering Next Generation Data Center Infrastructure. https://www.fungible.com/news/fungible-dpu-a-new-class-of-microprocessor-powering-next-generation-data-center-infrastructure/.

[4] Fungible F1 Data Processing Unit. https://www.fungible.com/wp-content/uploads/2020/08/PB0028.01.02020820-Fungible-F1-Data-Processing-Unit.pdf.

[5] Fungible S1 Data Processing Unit. https://www.fungible.com/wp-content/uploads/2020/08/PB0029.00.02020811-Fungible-S1-Data-Processing-Unit.pdf.

[6] Intel ethernet switch fm10000 datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-multi-host-controller-fm10000-family-datasheet.pdf.

[7] SHA-3 Hardware Accelerator. https://opencores.org/projects/sha3.

[8] Silicon at the speed of software. https://www.sifive.com. Accessed: 2020-05-25.

[9] Vexriscv. https://spinalhdl.github.io/SpinalDoc-RTD/SpinalHDL/Libraries/vexriscv.html.

[10] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[11] ACCOLADE TECHNOLOGY. Accolade ANIC. https://accoladetechnology.com/whitepapers/ANIC-Features-Overview.pdf.

[12] ALPHA DATA. ADM-PCIE-9V3 - High-Performance Network Accelerator. https://www.alpha-data.com/pdfs/adm-pcie-9v3.pdf.

[13] ARASHLOO, M. T., LAVROV, A., GHOBADI, M., REXFORD, J., WALKER, D., AND WENTZLAFF, D. Enabling programmable transport protocols in high-speed NICs. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2020).

[14] ASANOVIĆ, K., AVIZIENIS, R., BACHRACH, J., BEAMER, S., BIANCOLIN, D., CELIO, C., COOK, H., DABBELT, D., HAUSER, J., IZRAELEVITZ, A., KARANDIKAR, S., KELLER, B., KIM, D., KOENIG, J., LEE, Y., LOVE, E., MAAS, M., MAGYAR, A., MAO, H., MORETO, M., OU, A., PATTERSON, D. A., RICHARDS, B., SCHMIDT, C., TWIGG, S., VO, H., AND WATERMAN, A. The rocket chip generator. Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[15] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2011).

[16] BAREFOOT. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino, 2017.

[17] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[18] BERTONI, G., DAEMEN, J., PEETERS, M., AND ASSCHE, G. The keccak reference, version 3.0. *NIST SHA3 Submission Document (January 2011)* (2011).

[19] BERTONI, G., DAEMEN, J., PEETERS, M., AND VAN ASSCHE, G. Keccak sponge function family main document. *Submission to NIST (Round 2)* (2009).

[20] BHAGWAN, R., AND LIN, B. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)* (2000).

[21] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F. A., AND HOROWITZ, M. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2013).

[22] BROADCOM. Stingray SmartNIC Adapters and IC. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[23] CAVIUM CORPORATION. Cavium CN63XX-NIC10E. http://cavium.com/Intelligent_Network_Adapters_CN63XX_NIC10E.html.

[24] CAVIUM CORPORATION. Cavium LiquidIO. http://www.cavium.com/pdfFiles/LiquidIO_Server_Adapters_PB_Rev1.2.pdf.

[25] CELIO, C., CHIU, P.-F., ASANOVIĆ, K., NIKOLIĆ, B., AND PATTERSON, D. Broom: an open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos. *IEEE Micro* (2019).

[26] CELIO, C., CHIU, P.-F., NIKOLIC, B., PATTERSON, D., AND ASANOVIC, K. Boom v2, 2017.

[27] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., ORDA, A., AND EDSALL, T. dRMT: Disaggregated programmable switching. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2017).

[28] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., ORDA, A., AND EDSALL, T. dRMT: Disaggregated programmable switching - extended version. https://cs.nyu.edu/~anirudh/sigcomm17_drmt_extended.pdf, 2017.

[29] DAEMEN, J., AND RIJMEN, V. *The design of Rijndael: AES-the advanced encryption standard*. 2013.

[30] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[31] EXABLAZE. ExaNIC V5P. https://exablaze.com//exanic-v5p.

[32] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. Azure accelerated networking: SmartNICs in the public cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2018).

[33] FORENCICH, A., SNOEREN, A. C., PORTER, G., AND PAPEN, G. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines* (2020).

[34] GRANT, S., YELAM, A., BLAND, M., AND SNOEREN, A. C. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2020).

[35] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over commodity Ethernet at scale. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2016).

[36] HUMPHRIES, J. T., KAFFES, K., MAZIÈRES, D., AND KOZYRAKIS, C. Mind the Gap: A case for informed request scheduling at the NIC. In *ACM Workshop on Hot Topics in Networks (ACM HotNets)* (2019).

[37] IBANEZ, S., SHAHBAZ, M., AND MCKEOWN, N. The case for a network fast path to the CPU. In *ACM Workshop on Hot Topics in Networks (ACM HotNets)* (2019).

[38] INTEL. Intel 82599 10 GbE controller datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[39] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2015).

[40] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Datacenter rpcs can be general and fast. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[41] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., AND PORTER, G. Bullet trains: A study of NIC burst behavior at microsecond timescales. In *Conference on Emerging Networking Experiments and Technologies CoNEXT* (2013).

[42] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with FlexNIC. In *ASPLOS* (2016).

[43] KIM, J., DALLY, W. J., AND ABTS, D. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual International Symposium on Computer Architecture (ISCA)* (2007).

[44] KUON, I., AND ROSE, J. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays* (2006).

[45] LE, Y., CHANG, H., MUKHERJEE, S., WANG, L., AKELLA, A., SWIFT, M. M., AND LAKSHMAN, T. V. UNO: Uniflying host and smart NIC offload for flexible packet processing. In *SoCC* (2017).

[46] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *SOSP* (2017).

[47] LI, B., TAN, K., LUO, L., LUO, R., PENG, Y., XU, N., XIONG, Y., AND CHENG, P. ClickNP: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2016).

[48] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading distributed applications onto smartnics using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2019).

[49] LIU, M., PETER, S., KRISHNAMURTHY, A., AND PHOTHILIMTHANA, P. M. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *Usenix Annual Technical Conference (ATC)* (2019).

[50] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving resource efficiency at scale. In *International Symposium on Computer Architecture (ISCA)* (2015).

[51] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RYAN, M., RUBOW, E., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: a microkernel approach to host networking. In *SIGOPS* (2019).

[52] MELLANOX TECHNOLOGIES. Innova - 2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf.

[53] MELLANOX TECHNOLOGIES. Mellanox BlueField Smart-NIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[54] MELLANOX TECHNOLOGIES. NVIDIA Mellanox BlueField-2 DPU. https://www.mellanox.com/products/bluefield2-overview.

[55] MELLETTE, W. M., DAS, R., GUO, Y., MCGUINNESS, R., SNOEREN, A. C., AND PORTER, G. Expanding across time to deliver bandwidth efficiency and low latency. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2020).

[56] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[57] NETRONOME. NFP-6xxx flow processor. https://netronome.com/product/nfp-6xxx/.

[58] PENSANDO. DSC-100. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-100-Product-Brief.pdf.

[59] PHOTHILIMTHANA, P. M., LIU, M., KAUFMANN, A., PETER, S., BODIK, R., AND ANDERSON, T. Floem: A programming system for NIC-accelerated network applications. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[60] PONTARELLI, S., BIFULCO, R., BONOLA, M., CASCONE, C., SPAZIANI, M., BRUSCHI, V., SANVITO, D., SIRACUSANO, G., CAPONE, A., HONDA, M., HUICI, F., AND SIRACUSANO, G. FlowBlaze: Stateful packet processing in hardware. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[61] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2012).

[62] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for end-host rate limiting. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[63] SEWELL, K., DRESLINSKI, R. G., MANVILLE, T., SATPATHY, S., PINCKNEY, N., BLAKE, G., CIESLAK, M., DAS, R., WENISCH, T. F., SYLVESTER, D., ET AL. Swizzle-switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems 2*, 2 (2012), 278–294.

[64] SHRIVASTAV, V. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2019).

[65] SHU, R., CHENG, P., CHEN, G., GUO, Z., QU, L., XIONG, Y., CHIOU, D., AND MOSCIBRODA, T. Direct universal access: Making data center resources available to FPGA. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[66] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2015).

[67] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2016).

[68] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2016).

[69] STEPHENS, B., AKELLA, A., AND SWIFT, M. Your programmable NIC should be a programmable switch. In *ACM Workshop on Hot Topics in Networks (ACM HotNets)* (2018).

[70] STEPHENS, B., AKELLA, A., AND SWIFT, M. Loom: Flexible and efficient nic packet scheduling. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[71] THOMAS, S., MCGUINNESS, R., VOELKER, G. M., AND PORTER, G. Dark packets and the end of network scaling. In *ANCS* (2018).

[72] TILERA. Tile Processor Architecture Overview For the TILE-GX Series. http://www.mellanox.com/repository/solutions/tile-scm/docs/UG130-ArchOverview-TILE-Gx.pdf.

[73] WENTZLAFF, D., GRIFFIN, P., HOFFMANN, H., BAO, L., EDWARDS, B., RAMEY, C., MATTINA, M., MIAO, C.-C., BROWN III, J. F., AND AGARWAL, A. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 5 (Sept. 2007).

[74] WILTON, S. J. E., HO, C. H., LEONG, P. H. W., LUK, W., AND QUINTON, B. A synthesizable datapath-oriented embedded FPGA fabric. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA)* (2007).

[75] XILINX. Xilinx Alveo: Adaptable Accelerator Cards for Data Center Workloads. https://www.xilinx.com/products/boards-and-kits/alveo.html.

[76] ZILBERMAN, N., AUDZEVICH, Y., COVINGTON, G., AND MOORE, A. NetFPGA SUME: Toward 100 Gbps as research commodity.

A Artifact Appendix

A.1 Abstract

This artifact contains the source code and test benches for PANIC's 100Gbps FPGA-based prototype. Our FPGA prototype is implemented in pure Verilog. Features of the prototype include: the hybrid push/pull packet scheduler, the high-performance switching interconnect, self-contained compute units, and the lightweight RMT pipeline.

This artifact provides two test benches to reproduce the results in Figure 8c and Figure 11a in the Vivado HDL simulator.

A.2 Artifact check-list

- **Compilation:** Running this artifact requires Vivado Design Suite [10]. Vivado v2019.x and v2020.1 WebPack are verified.
- **Hardware:** This artifact does not requires any specific hardware.
- **Metrics:** This artifact measures PANIC's receiving throughput under different chaining models and traffic patterns.
- **Output:** The result will be printed to the console and log files.
- **Experiments:** This artifact includes testbenches and running scripts to replay Figure 8c and Figure 11a.
- **Public link:** https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact

A.3 Description

A.3.1 How to access

This artifact is publicly available at https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact.

A.3.2 Software dependencies

Running this artifact requires Vivado [10]. Vivado WebPack version is license-free, and it has simulation capabilities to recreate our results. Since installing the Vivado WebPack requires plenty of disk space (>20GB), you can choose to instance an FPGA Developer AMI in AWS (https://aws.amazon.com/marketplace/pp/B06VVYBLZZ) to run this artifact. The FPGA Developer AMI has pre-installed the required Vivado toolchain.

A.4 Experiment workflow

1. **Check Vivado is Installed Correctly**

```
$ vivado -mode tcl
// Enter the Vivado Command Palette
Vivado% version
// v2019.x and v2020.1 is verified
Vivado% quit
```

2. **Clone the Repo and Make Run**

```
$ git clone [Artifact_Repo]
$ cd panic_osdi20_artifact
$ make test_parallel
$ make test_shaaes
```

The make command first compiles the source code, then runs the simulation tasks in Vivado. The *test_parallel* test replays Figure 8c and the *test_shaaes* test replays Figure 11a.

A.5 Evaluation and expected result

The result will be printed to the console. The output will also be logged in *./build/export_sim/xsim/simulate.log*. For the expected output and analysis please reference Figure 8c and Figure 11a.

A.6 Notes

For more details about the code structure, please reference https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact/src/master/README.md

A.7 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts





Semeru: A Memory-Disaggregated Managed Runtime

Chenxi Wang[†] Haoran Ma[†] Shi Liu[†] Yuanqi Li[†] Zhenyuan Ruan[‡] Khanh Nguyen[§]
Michael D. Bond[*] Ravi Netravali[†] Miryung Kim[†] Guoqing Harry Xu[†]
UCLA[†] *MIT*[‡] *Texas A&M University*[§] *Ohio State University*[*]

Abstract

Resource-disaggregated architectures have risen in popularity for large datacenters. However, prior disaggregation systems are designed for native applications; in addition, all of them require applications to possess excellent locality to be efficiently executed. In contrast, programs written in managed languages are subject to periodic garbage collection (GC), which is a typical graph workload with poor locality. Although most datacenter applications are written in managed languages, current systems are far from delivering acceptable performance for these applications.

This paper presents *Semeru*, a distributed JVM that can dramatically improve the performance of managed cloud applications in a memory-disaggregated environment. Its design possesses three major innovations: (1) a universal Java heap, which provides a unified abstraction of virtual memory across CPU and memory servers and allows any legacy program to run *without modifications*; (2) a distributed GC, which offloads *object tracing* to memory servers so that tracing is performed *closer to data*; and (3) a swap system in the OS kernel that works with the runtime to swap page data efficiently. An evaluation of *Semeru* on a set of widely-deployed systems shows very promising results.

1 Introduction

The idea of *resource disaggregation* has recently attracted a great deal of attention in both academia [16, 45, 49, 87] and industry [3, 33, 39, 52, 65]. Unlike conventional datacenters that are built with *monolithic* servers, each of which tightly integrates a small amount of each type of resource (*e.g.*, CPU, memory, and storage), resource-disaggregated datacenters contain servers dedicated to individual resource types. Disaggregation is particularly appealing due to three major advantages it provides: (1) *improved resource utilization*: decoupling resources and making them accessible to remote processes make it much easier for a job scheduler to achieve full resource utilization; (2) *improved failure isolation*: any server failure only reduces the amount of resources of a particular type, without affecting the availability of other types of resources; and (3) *improved elasticity*: hardware-dedicated servers make it easy to adopt and add new hardware.

State of the Art. Architecture [10, 22, 23, 58] and networking [7, 30, 46, 55, 72, 83, 86, 88] technologies have matured to a point at which data transfer between servers is fast enough for them to execute programs collectively. LegoOS [87] provides a new OS model called *splitkernel*, which disseminates traditional OS components into loosely coupled monitors, each of which runs on a resource server. InfiniSwap [49] is a paging system that leverages RDMA to expose memory to applications running on remote machines. FaRM [37] is a distributed memory system that uses RDMA for both fast messaging and data access. There also exists a body of work [12, 28, 38, 60, 61, 64, 65, 73, 77, 94, 96, 97, 105] on storage disaggregation.

1.1 Problems

Although RDMA provides efficient data access among remote access techniques, fetching data from remote memory on a memory-disaggregated architecture, is time consuming, incurring microsecond-level latency that cannot be handled well by current system techniques [20]. While various optimizations [37, 38, 49, 84, 87, 105] have been proposed to reduce or hide fetching latency, such techniques focus on the low-level system stack and do *not* consider *run-time semantics* of a program, such as locality.

Improving performance for applications that exhibit *good locality* is straightforward: the CPU server runs the program, while data are located on memory servers; the CPU server has only a small amount of memory used as a *local cache*[1] that stores recently fetched pages. A cache miss triggers a page fault on the CPU server, making it fetch data from the memory server that hosts the requested page. Good locality reduces cache misses, leading to improved application performance. As a result, a program itself needs to possess *excellent spatial and/or temporal locality* to be executed efficiently under current memory-disaggregation systems [7, 8, 49, 87].

This high requirement of locality creates two practical challenges for cloud applications. First, typical cloud applications are written in managed languages that execute atop a managed runtime. The runtime performs automated memory management using *garbage collection (GC)*, which frequently traces the heap and reclaims unreachable objects. GC is a typical graph workload that performs reachability analysis over a huge graph of objects connected by references. Graph traversal often suffers from poor locality, so GC running on the CPU server potentially triggers a page fault as it follows each reference. As shown in §2, memory disaggregation can increase the duration of GC pauses by >10×, significantly degrading application performance.

[1] In this paper, "cache" refers to local memory on the CPU server.

Second, to make matters worse, unlike native programs whose data structures are primarily array-based, managed programs make heavy use of object-oriented data structures [74, 100, 101], such as maps and lists connected via pointers without good locality. To illustrate, consider a Spark RDD — it is essentially a large list that references a huge number of element objects, which can be distributed across memory servers. Even a sequential scan of the list needs to access arbitrarily located elements, incurring high performance penalties due to frequent remote fetches.

In essence, managed programs such as Spark, which are typical cloud workloads that resource disaggregation aims to benefit, have not yet received much support from existing resource-disaggregated systems.

1.2 Our Contributions

Goal and Insight. The goal of this project is to design a memory-disaggregation-friendly managed runtime that can provide superior efficiency to all managed cloud applications running in a memory-disaggregated datacenter. Our major drive is an observation that shifting our focus from low-level, semantics-agnostic optimizations (as done in prior work) to the *redesign of the runtime* that improves data placement, layout, and usage, can unlock massive opportunities.

To achieve this goal, our insights are as follows. To exploit *locality for GC*, most GC tasks can be *offloaded* to memory servers where data is located. As GC tasks are mostly memory intensive, this offloading fits well into a memory server's resource profile: weak compute and abundant memory. Memory servers can perform some offloaded GC tasks — such as tracing objects — *concurrently* with application execution. Similarly, other GC tasks — such as evacuating objects and reclaiming memory — can be offloaded to memory servers, albeit while application execution is paused. Furthermore, evacuation can improve *application locality* by moving objects likely to be accessed together to contiguous memory.

***Semeru*.** Following these insights, we develop *Semeru*,[2] a distributed Java Virtual Machine (JVM) that supports efficient execution of *unmodified* managed applications. As with prior work [49, 87], this paper assumes a setting where processes on each CPU server can use memory from multiple memory servers, but no single process spans multiple CPU servers. *Semeru*'s design sees three major challenges:

The first challenge is what memory abstraction to provide. A reachability analysis over objects on a memory server requires the server to run a user-space process (such as a JVM) that has its own address space. As such, the same object may have different virtual addresses between the CPU server (that runs the main process) and its hosting memory server (that runs the tracing process). Address translation for each object can incur large overheads.

To overcome this challenge, *Semeru* provides a memory abstraction called the *universal Java heap (UJH)* (§3.1). The execution of the program has a main compute process running on the CPU server as well as a set of "assistant" processes, each running on a memory server. The main and assistant processes are all JVM instances, and servers are connected with RDMA over InfiniBand. The main process executes the program while each assistant process only runs offloaded memory management tasks. The heap of the main process sees a contiguous virtual address space partitioned across the participating memory servers, each of which sees and manages a disjoint range of the address space. *Semeru* enables an object to have the same virtual address on both the CPU server and its hosting memory server, making it easy to separate an application execution from the GC tasks.

The second challenge is what to offload. An ideal approach is to run the entire GC on memory servers while the CPU server executes the program, so that memory management tasks are performed (1) near data, providing locality benefits, and (2) concurrently without interrupting the main execution. However, this approach is problematic because some GC operations — notably evacuating (moving) and compacting objects into a new region — must coordinate extensively with application threads to preserve correctness. As a result, many GC algorithms — including the high-performance GC that our work extends — trace live objects concurrently with application execution, but move objects only while application execution is paused (*i.e.*, stop-the-world collection).

We develop a distributed GC (§4) that selectively offloads tasks and carefully coordinates them to maximize GC performance. Our idea is to offload *tracing* to memory servers concurrently with application execution. Tracing computes a transitive closure of live objects from a set of roots. It does nothing but pointer chasing, which would be a major bottleneck if performed at the CPU server. To avoid this bottleneck, *Semeru* lets each memory server trace its own objects, as opposed to bringing them into the CPU server for tracing.

Tracing is a memory-intensive task that does not need much compute [27] but benefits greatly from being close to data. To leverage memory servers' weak compute, memory servers trace their local objects *continuously* while the CPU server executes the main threads. Tracing also fits well into various hardware accelerators [69, 85], which future memory servers may employ. The CPU server periodically stops the world for memory servers to *evacuate* live objects (*i.e.*, copy them from old to new memory regions) to reclaim memory. Object evacuation provides a unique opportunity for *Semeru* to relocate objects that may potentially be accessed together into a *contiguous* space, improving spatial locality.

The third challenge is how to efficiently swap data. Existing swap systems such as InfiniSwap [49] and FastSwap [11] cannot coordinate with the language runtime and have bugs when running distributed frameworks such as Spark (§2). Mellanox provides an NVMe-over-fabric (NVMe-oF) [1] driver that allows the CPU server to efficiently access remote storage using RDMA. A strawman approach here is to mount

[2] *Semeru* is the highest mountain on the island of East Java.

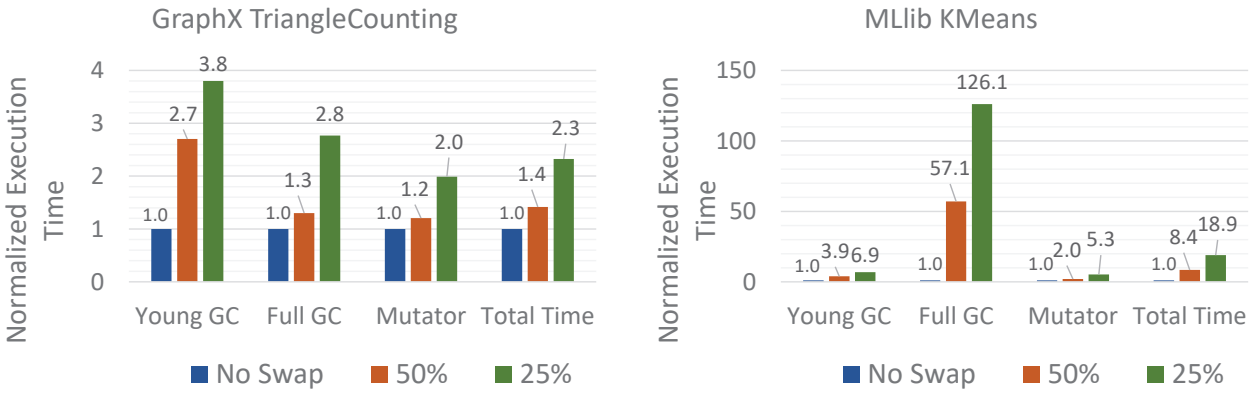


Figure 1: Slowdowns of two representative Spark applications under disaggregated memory; NVMe-oF was used for data swapping. Spark was executed over OpenJDK 12 with its default (Garbage First) GC. The four groups for each program report the slowdowns of the nursery (young) GC, full-heap GC, mutator, and end-to-end execution. Each group contains three bars, reporting the execution times under three cache configurations: 100%, 50%, and 25%. Each configuration represents a percentage of the application's working set that can fit into the CPU server's local DRAM. Execution times of the 50% and 25% configurations are normalized to that of 100%.

remote memory as RAMDisks and use NVMe-oF to swap data. However, this approach does not work in our setting where remote memory is subject to memory-server tracing and compaction, precluding it from being used as RAMDisks. To this end, we modify the NVMe-oF implementation (§5) to provide support for remote memory management. InfiniBand gather/scatter is used to efficiently transfer pages. We also develop new system calls that enable effective communications between the runtime and the swap system.

Results. We have evaluated *Semeru* using two widely-deployed systems – Spark and Flink – each with a representative set of programs. Our results demonstrate that *Semeru* improves the end-to-end performance of these systems by an average of **2.1**× and **3.7**× when the cache size is 50% and 25% of the heap size, application performance by an average of **1.9**× and **3.3**×, and GC performance by **4.2**× and **5.6**×, respectively, compared to running these systems directly on NVM-oF where remote accesses incur significant latency overheads. These promising results suggest that *Semeru* reduces the gap between memory disaggregation and managed cloud applications, taking a significant step toward efficiently running such applications on disaggregated datacenters.

Semeru is publicly available at `https://github.com/uclasystem/Semeru`.

2 Motivation

We conducted experiments to understand the latency penalties that managed programs incur on existing disaggregation systems. We first tried to use existing disaggregation systems including LegoOS [87], InfiniSwap [49], and FastSwap [11]. However, LegoOS does not yet support socket system calls and cannot run socket-based distributed systems such as Spark. Under InfiniSwap and FastSwap, the JVM was frequently stuck — certain remote fetches never returned.

Background of G1 GC. To collect preliminary data, we set up a small cluster with one CPU and two memory servers, using Mellanox's NVMe-over-fabric (NVMe-oF) [1] protocol for data swapping, mounting remote memory as a RAMDisk. On this cluster, we ran two representative Spark applications: Triangle Counting (TC) from GraphX and KMeans from MLib with the Twitter graph [63] as the input. We used OpenJDK 12 with its high-performance *Garbage First (G1)* GC, which is the default GC recommended for large-scale processing tasks, with a 32GB heap. G1 is a *region-based, generational* GC that most frequently traces the young generation (*i.e.*, nursery GC) and occasionally traces both young and old generations (*i.e.*, full-heap GC). This is based on the *generational hypothesis* that most objects die young and hence the young generation contains a larger fraction of garbage than the old generation [93].

Under G1, the memory for both the young and old generations is divided into *regions*, each being a contiguous range of address space. Objects are allocated into regions. Each nursery GC traces a small number of selected regions in the young generation. After tracing, live objects in these regions are evacuated (*i.e.*, moved) into new regions. Objects that have survived a number of nursery GCs will be *promoted* to the old generation and subject to less frequent tracing. Each full-heap GC traces the entire heap, and then evacuates and compacts a subset of regions.

Performance. The performance of these applications is reported in Figure 1. In particular, we measured time spent on nursery and full-heap collections, as well as end-to-end execution time. Three cache configurations (shown in three bars of each group) were considered, each representing a particular percentage of the application's working set that can fit into the CPU server's local DRAM.

Despite the many block-layer optimizations in the NVMe-oF swap system, performance penalties from remote fetching are still large. Under the 25% cache configuration, the average slowdown for these applications is **10.6**×. Note that for a typical Big Data application with a large working set (*e.g.*, 80–100GB), 25% of the working set means that the CPU server

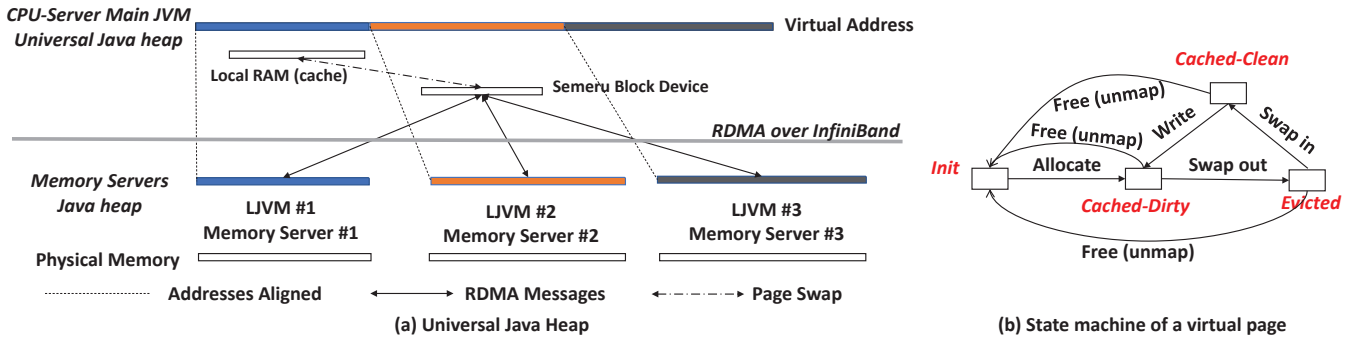


Figure 2: *Semeru*'s heap and virtual page management.

needs at least 20–25GB DRAM for *a single application* to have a ∼10× slowdown. Considering a realistic setting where the CPU server runs multiple applications, there is a much higher DRAM requirement for the CPU server, posing a practical challenge for disaggregation.

Takeaway. Disaggregated memory incurs a higher slowdown for the GC than the main application threads (*i.e.*, *mutator* threads in GC literature terminology) — this is easy to understand because compared to the mutator (which, for example, manipulates large Spark RDD arrays), the GC has much worse locality. Moreover, KMeans suffers much more from remote memory than TC due to significantly increased full-heap GC time. This is because KMeans uses a number of persisted RDDs (that are held in memory indefinitely). Although TC also persists RDDs, those RDDs are too large to be held in memory; as such, Spark releases them and reconstructs them when they are needed. This increases the amount of computation but reduces the GC effort under disaggregation. However, since memoization is an important and widely used optimization, it is not uncommon for data processing applications to hold large amounts of data in memory. As a result, these applications are expected to suffer from large-working-set GC as well.

These results call for a new managed runtime that can deliver good performance under disaggregated memory without requiring developers to be aware of and reason about the effects of disaggregation during development.

3 *Semeru* Heap and Allocator

This section discusses the design of *Semeru*'s memory abstraction. In order to support legacy applications developed for monolithic servers and to hide the complexity of data movement, we propose the *universal Java heap (UJH)* memory abstraction. We first describe this abstraction, and then discuss object allocation and management.

3.1 Universal Java Heap

The main process (*i.e.*, a JVM instance) running on the CPU server sees a large contiguous virtual address space, which we refer to as the universal Java heap. The application can access any part of the heap regardless of the physical locations. This contiguous address space is *partitioned* across memory servers, each of which provides physical memory that backs a disjoint region of the universal heap. The CPU server also has a small amount of memory, but this memory will serve as a software-managed, inclusive cache and hence not be dedicated to specific virtual addresses. Mutator (*i.e.*, application) threads run on the CPU server. When they access pages that are uncached on the CPU server, a page fault is triggered, and the paging system swaps pages that contain needed objects into the CPU server's local memory (cache). When the cache is full, selected pages are swapped out (evicted) to their corresponding memory servers, as determined by their virtual addresses.

Figure 2(a) provides an overview of the UJH. In addition to the main process running on the CPU server, *Semeru* also runs a *lightweight* JVM (LJVM) process on each participating memory server that performs tracing over local objects. This LJVM[3] is specially crafted to contain only the modules of object tracing and memory compaction, with support for RDMA-enabled communication with the CPU server. Due to its simplicity (*i.e.*, the modules of compiler, class loader, and runtime as well as much of the GC are all eliminated), the LJVM has a very short initialization time (*e.g.*, milliseconds) and low memory footprint (*e.g.*, megabytes of memory for tracing metadata). Hence, a memory server can easily run many LJVMs despite its weak compute (*i.e.*, each for a different CPU-server process).

When the LJVM starts, it aligns the starting address of its local heap with that of its corresponding address range in the UJH. As a result, each object has the same virtual address on the CPU and memory servers, enabling memory servers to trace their local objects without address translation. All physical memory required at each memory server is allocated when the LJVM is launched and pinned down during the entire execution of the program.

Coherency. This memory abstraction is similar in spirit to distributed shared memory (DSM) [66], which has been studied for decades. However, different from DSM, which needs to provide strong coherency between servers, *Semeru*'s coherency protocol is much simpler because memory servers, which collectively manage the address space, do not execute

[3] It is technically no longer a JVM since it does not execute Java programs.

any mutator code. The CPU server has access to the entire UJH, but each memory server can only access data in the address range it manages. In *Semeru*, each non-empty virtual page is in one of two high-level states, *cached* (in the CPU server) or *evicted* (to a memory server). When the CPU server accesses an *evicted* virtual page, it swaps the page data into its cache and changes the page's state to *cached*.

3.2 Allocation and Cache Management

Object allocation is performed at the CPU server. Allocation finds a virtual space that is large enough to accommodate the object being allocated. We adopt G1's region-based heap design where the heap is divided into *regions*, which are contiguous segments of virtual memory. The region-based design enables *modular tracing and reclamation* — each memory server hosts a set of regions; a memory server can trace any region it hosts independently of other regions, thereby enabling memory servers to perform tracing in parallel (while the CPU server executes the program). Modular tracing is enabled by using *remembered sets*, discussed shortly in §4.

When an object in a region is requested by the CPU server, the page(s) containing the object are swapped in. At this point, the region is *partially cached* and registered at the CPU server into an *active region list*. *Semeru* uses a simple LRU-based cache management algorithm to evict pages. The region is removed from this list whenever all its pages are evicted.

Upon an allocation request, the *Semeru* allocator finds the first region from this list that has enough space for the new object. If none of these regions can satisfy the request, *Semeru* creates a new region and allocates the object there. Allocation is based upon an efficient *bump pointer* algorithm [57], which places allocated objects contiguously and in allocation order. Bump pointer allocation maintains a position pointer for each region, pointing to the starting address of the free space. Bump pointer allocation maintains a position pointer for each region that points to the starting address of the region's free space. For each allocation, the pointer is simply "bumped up" by the size of the allocated object. Very large objects are allocated to a special heap area called the *humongous space*.

```
1 struct region {
2   uint64_t start;        // start address
3   uint64_t bp;           // bump pointer
4   uint64_t num_obj;      // total # objects
5   uint64_t cached_size; // size of pages in CPU cache
6   uint16_t survivals;    // # evacuations survived
7   remset* rem_set;       // remembered set (Section 4)
8   ...
9 }
```

Figure 3: A simplified definition for a region descriptor in *Semeru*.

The CPU server maintains, for all regions, their state *descriptors*. Each region descriptor is a `struct`, illustrated in Figure 3. Descriptors are used in both allocation and garbage collection. For example, `start` and `bp` are used for allocation; they can also be used to calculate the size of allocated objects. `survivals` indicates the total number of evacuation phases that the regions' objects have survived. It can be used, together with `num_obj`, to compute an *age* measurement for the region. `rem_set` is used as the tracing roots, which will be discussed shortly in §4.2.

Cache Management. *Semeru* employs a lazy write-back technique for allocations. Each allocated object stays in the CPU server's cache and *Semeru* does not write the object back to its corresponding memory server until the pages containing the object are evicted. For efficiency, only dirty pages are written back. Figure 2(b) shows the state machine of a virtual page. Each virtual page is initially in the `Init` state. Upon an object allocation on a page, the object is placed in the cache of the CPU server and its virtual page is marked as `Cached`, indicating that the object is currently being accessed by the CPU server. Evicted pages are swapped out to memory servers. Virtual pages freed by the GC are *unmapped* from their physical pages (their corresponding page table entries are *not* freed) and have their states reset to `Init`. This state machine is managed solely by the CPU server; memory servers do not run application code and hence do not need to know the state of each *page* (although they need to know the state of *regions* for tracing).

4 *Semeru* Distributed Garbage Collector

Semeru has a distributed GC that offloads tracing — the most memory-intensive operation in the GC (as it visits every live object) — to memory servers. Tracing is a task that fits well into the capabilities of a memory server with limited compute. That is, traversing an object graph by chasing pointers does not need strong compute, but benefits greatly from being close to data. In addition to memory-server tracing that runs continuously, *Semeru* periodically conducts a highly parallel stop-the-world (STW) collection phase to free cache space on the CPU server and reclaim memory on memory servers by evacuating live objects.

Design Overview. Although regions have been used in prior heap designs [36, 79], there are two unique challenges in using regions efficiently for disaggregated memory.

The first challenge is *how to enable modular tracing for regions.* Prior work such as Yak [79] builds a *remembered set* (remset) for each region that records references coming into objects in the region from other regions. These references, which are recorded into the set by instrumentation code called a *write barrier*, when the mutator executes each *object write of a non-null reference value*, can be used as additional *roots* to traverse the object graph for the region. However, none of the existing techniques consider a distributed scenario, where region tracing is done on memory servers, while their remsets are updated by mutator threads on the CPU server. We propose a new distributed design of the remset data structure to minimize the communication between the CPU and memory servers. Our remset design is discussed in §4.1.

The second challenge is *how to split the GC tasks between servers.* Our distributed GC has two types of collections:

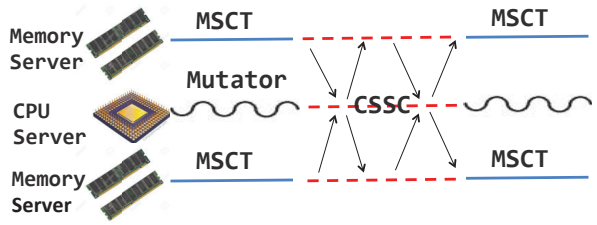


Figure 4: *Semeru* GC overview: the MSCT (on memory servers) traces evicted regions; the CSSC (coordinated between CPU and memory servers) traces cached regions and reclaims all regions.

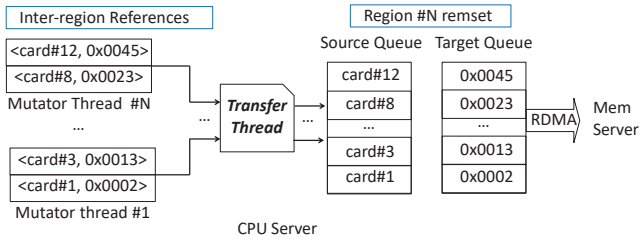


Figure 5: *Semeru*'s remset design; the source and target queues are implemented as bitmaps for space efficiency.

Memory Server Concurrent Tracing (MSCT, §4.2): Each memory server performs *intra-region* tracing over *regions for which most pages are evicted*, as a *continuous task*. Tracing runs *concurrently* on memory servers by leveraging their cheap but idle CPU resources. One can think of this as a background task that does not add any overhead to the application execution. The goal of MSCT is to compute a live object closure for each region at memory servers without interfering with the main execution at the CPU server. As a result, by the time a STW phase (*i.e.*, CSSC) runs, much of the tracing work is done, minimizing the STW pauses.

CPU Server Stop-The-World Collection (CSSC, §4.3): The CSSC is the main collection phase, coordinated between the CPU and memory servers to reclaim memory. During this phase, memory servers follow the per-region object closure computed during the MSCT to evacuate (*i.e.*, move out) live objects. Old regions are then reclaimed as a whole. Also during this phase, the CPU server traces and reclaims regions *for which most pages are cached*. Such regions are not traced by the MSCT. For evacuated objects, pointers pointing to them need to be updated in this phase as well.

Figure 4 shows an overview of these two types of collections. While the CPU server runs mutator threads, memory servers run the MSCT that continuously traces their hosted regions. When the CPU server stops the world and runs the CSSC, memory servers suspend the MSCT and coordinate with the CPU server to reclaim memory.

4.1 Design of the Remembered Set

The remset is a data structure that records, for each region, the references coming into the region. The design of the remset is much more complicated under a memory-disaggregated architecture due to the following two challenges. First, in a traditional setting, to represent an inter-region reference (*e.g.*, from field $o.f$ to object p), we only need its *source* location — the address of $o.f$. This is because p can be easily obtained by following the reference in $o.f$. However, in our setting, both $o.f$ and p need to be recorded for efficiency. This is because o and p can be on different servers and naïvely following the reference in $o.f$ can trigger a remote access.

The second challenge is that the remset of each region is updated by the write barrier executed on the CPU server, while the region may be traced by a memory server. As a result, the CPU server has to periodically send the remsets to memory servers for them to concurrently trace their regions. In addition, after memory servers evacuate objects, they need to send update addresses for the remsets back to the CPU server for it to update the sources of references (*e.g.*, $o.f$ may point to a moved object p).

Figure 5 shows our remset. To represent the source of a reference, we leverage OpenJDK's *card table*, which groups objects into fixed-sized buckets (*i.e.*, cards) and tracks which buckets contain references. A card's ID can be easily computed (*i.e.*, via a bit shift) from a memory address and yet we can enjoy the many space optimizations already implemented in OpenJDK (*e.g.*, for references on *hot* cards that contain references going to the same region [36], their sources need to be recorded only once). As such, each incoming reference is represented as a pair $\langle$`card`, `tgt`$\rangle$ where `card` is the (8-byte) index of the card representing the source location of the reference, and `tgt` is the (8-byte) address of the target object.

Shown on the left side of Figure 5 are inter-region references recorded by the write barrier of each mutator thread. To reduce synchronization costs, each mutator thread maintains a thread-local queue storing its own inter-region references. The CPU-server JVM runs a daemon (transfer) thread that periodically moves these references into the remsets of their corresponding regions (*i.e.*, determined by the target addresses). For each region, a pointer to its remset is saved in the region's descriptor (Figure 3), which can be used to retrieve the remset by the CPU server. When a reference is recorded in a remset, its `card` and `tgt` are decoupled and placed separately into a source and a target queue.

Target queues are sent (together with stack references) — during each CSSC via RDMA — to their corresponding memory servers, which use them as roots to compute a closure over live objects. Source queues stay on the CPU server and are used during each CSST to update references if their target objects are moved during evacuation. The benefit of using a transfer thread is that mutator threads simply dump inter-region references, while the work of separating sources and targets and deduplicating queues (based on a simple hash-based data structure) is done by the transfer thread, which does not incur overhead on the main (application) execution.

4.2 Memory Server Concurrent Tracing (MSCT)

The MSCT brings significant efficiency benefits because (1) tracing computation runs where data is located, avoiding

high swapping costs, and (2) tracing regions *concurrently* on multiple memory servers has zero impact on the execution of the main application on the CPU server.

The MSCT *continuously* traces regions (until the CSSC starts) in the order of a region's age (*i.e.*, the smaller the value of `survivals`, the younger a region) and the percentage of evicted pages. That is, younger regions with more evicted pages are traced earlier. This is because (1) younger regions are likely to contain more garbage (according to the generational hypothesis), and (2) evicted pages are not touched by the CPU server. Regions with a low ratio of evicted pages are *not* traced since cached objects may be frequently updated by the CPU server. Tracing such regions would be less profitable because these updates can change pointer structures frequently, making the tracing results stale.

Identifying Roots. There are two types of roots for the MSCT to trace a region: (1) objects referenced by stack variables and (2) cross-region references recorded in the region's remset. Both types of information come from the CPU server — during each CSSC (§4.3), the CPU server scans its stacks, identifies objects referenced by stack variables, and sends this information, together with each region's remset, to its corresponding memory server via RDMA.

Live Object Marking. The MSCT computes a closure of reachable objects in each region by traversing the object subgraph (within the region) from its roots. When live objects are traversed, we remember them in a per-region bitmap `live_bitmap` where each bit represents a contiguous range of 8 bytes (because the size of an object is always a multiple of 8 bytes), and the bit is set if these bytes host a live object. Furthermore, since live objects will be eventually evacuated, we compute a new address for a live object as soon as it is marked. The new address indicates where this object will be moved to during evacuation. New addresses are recorded in a *forward table* (*i.e.*, a key–value store) where keys are the indexes of the set bits in `live_bitmap` and values are the new addresses of the live objects represented by these bits.

Each new address is represented as an *offset*. At the start of the MSCT, it is unclear where these objects will be moved to (since evacuation will not be performed until a CSSC). As a result, rather than using absolute addresses, we use offsets to represent their relative locations. Their actual addresses can be easily computed using these offsets once the starting address of the destination space is determined.

Offset computation is in *traversal order*. For example, the first object reached in the graph traversal receives an offset 0; the offset for the second object is the size of the first object. This approach dictates that *objects that are contiguous in traversal will be relocated to contiguous space after evacuation*. Hence, the traversal order, which determines which objects will be *contiguously placed* after evacuation, is critical for improving data locality and prefetching effectiveness.

For instance, if the traversal algorithm uses DFS, *objects connected by pointers* will be relocated to contiguous memory (based on an observation that such objects are likely in the same logical data structure and hence accessed contiguously). As another example, if we use BFS to traverse the graph, *objects at the same level of a data structure* (such as elements of an array) will be relocated to contiguous memory; this can be useful for streaming applications that may do a quick linear scan of all such element objects (*i.e.*, BFS) rather than fully exploring each element (*i.e.*, DFS). To support these different heuristics, *Semeru* allows the user to customize the traversal algorithm for different workloads.

Tracing Correctness. There are two potential concerns in tracing safety. First, if a region has a cached page, can the memory server safely trace the region (given that the CPU server may update the cached page)? For example, if an update happens after tracing completes, would the tracing results still be valid? Second, the root information may be out of date when a region is traced because the CPU server may have updated certain inter-region references or stack variables since the previous CSSC (where roots are computed and sent). Is it safe to trace with such out-of-date roots?

The answer to both questions is that it is still valid for a memory server to trace a region over an *out-of-date* object graph. An important safety property is that *objects unreachable in any snapshot of the object graph will remain unreachable in any future snapshots* (*i.e.*, "once garbage, always garbage"). Thus the transitive closure may include dead objects (due to pointer changes the memory server is not aware of), but objects *not* in the closure are guaranteed to be dead (except for newly allocated objects, discussed next).

However, tracing using an out-of-date object graph may lead to two issues. First, the CPU server may allocate new objects into a region *after* the region is traced on a memory server. These new objects are missed by the closure computation. To solve this problem, we identify *all* objects that have been allocated into the region *since the last CSSC*; such objects are all marked live at the time the region is reclaimed in the next CSSC so that no live object is missed. Newly allocated objects can be identified by remembering the value of the bump pointer (`bp` in Figure 3) at the the last CSSC and comparing it with the current value of `bp` — the difference between them captures objects allocated since the last CSSC. Such handling is *conservative*, because some of the objects may be dead already but are still included in the closure.

The second issue is that some objects in the region may lose their references and become unreachable after tracing is done. These dead objects are still in the closure. For this issue, we take a *passive* approach by not doing anything — we simply let these dead objects stay in the closure and be moved during evacuation. These dead objects will be identified in that next MSCT and collected during the next CSSC. Essentially, we delay the collection of these objects by one CSSC cycle. Note that datacenter applications are often not resource strapped; hence, delaying memory reclamation by one GC cycle is a better choice than an aggressive alternative that retraces the

region before reclamation (which can increase the length of each CSSC pause).

Handling CPU Evictions. A significant challenge is that concurrent tracing of a region can potentially *race* with the CPU server evicting a page into the region. To complicate matters, memory servers are not aware of remote reads/writes due to *Semeru*'s use of one-sided RDMA (for efficiency). Although recent RDMA libraries (such as LITE [91]) provide rich synchronization support, our use of RDMA at the block layer has many specific needs that are not met by these libraries, which were developed for user-space applications.

To overcome this challenge, we develop a simple workaround: each memory server reserves the *first 4 bytes of each region* to store two tags $\langle dirty, ver \rangle$. The first 2 bytes encode a boolean *dirty* tag and the second 2 bytes encode an integer *version* tag. These two tags are updated by the CPU server both before and after evicting pages into a region, and checked by the memory server both before and after the region is traced. Figure 6 shows this logic.

```
|-- 16-bit Dirty tag   --
|-- 16-bit Version tag --|

1 atomic_write(<1, v1>);
2 evict_pages();
3 atomic_write(<0, v2>);
```

(a) CPU server eviction

```
4  <dirty, ver> = atomic_read();
5  if(!dirty) {
6    trace();
7    <dirty, ver1> = atomic_read();
8    if(ver != ver1) discard();
9  }
10 else skip();
```

(b) MSCT tracing

Figure 6: Detection of evictions at a memory server.

Before evicting pages, the CPU server assigns 1 to the dirty tag and a new version number v_1 to the version tag (Line 1). This 4-byte information is written atomically by the RDMA network interface controller (RNIC) into the target region. After eviction, the CPU server clears the dirty tag and writes another version number v_2 (Line 3). The memory server reads these 4 bytes atomically and checks the dirty tag (Line 4). If it is set, this indicates a potential eviction; the memory server skips this region and moves on to tracing the next region (Line 10). Otherwise, the region is traced (Line 6). After tracing, this metadata is retrieved again and the new version tag is compared with the pre-tracing version tag. A difference means that an eviction may have occurred and the tracing results are discarded (Line 8).

The algorithm is sufficient to catch all concurrent evictions. The correctness can be easily seen by reasoning about the following three cases. (1) If Line 1 comes before Line 4 (which comes before Line 3), tracing will not be performed. (2) If Line 1 comes after Line 4 but before Line 8, the version check at Line 8 will fail. (3) If Line 1 comes after Line 7, the eviction has no overlap with the tracing and thus the tracing results are legitimate.

This algorithm introduces overheads due to extra write-backs. However, by batching pages from the same region and employing InfiniSwap's gather/scatter, we manage to reduce this overhead to about 5%, which can be easily offset by the savings achieved by tracing objects on memory servers (see §6.4). Concurrent CPU-server reads are *allowed*. Similar to tracing out-of-date object graphs, fetching a page into the CPU server can potentially lead to new objects and pointer updates to the page. However, our aforementioned handling is sufficient to cope with such scenarios.

4.3 CPU Server Stop-The-World Collection (CSSC)

CSSC Overview. As the major collection effort, the CSSC runs when (1) the heap usage exceeds a threshold, *e.g.*, $N\%$ of the heap size, or (2) *Semeru* observes large amounts of swapping. The CPU server suspends all mutator threads and collaborates with memory servers to perform a collection. Our goal is to (1) reclaim cache memory at the CPU server and (2) provide a STW phase for memory servers to safely reclaim memory by evacuating live objects in the traced regions. Figure 7 overviews the CSSC protocol; edges represent communications of GC metadata between CPU and memory servers. The CSSC has four major tasks.

Task 1: The CPU server prepares information for memory servers to reclaim regions. Such information includes which regions to reclaim at each memory server (❶) and newly allocated objects for each region to be reclaimed (❷). As discussed in §4.2, newly allocated objects need to be marked live for safety and are identified by differencing the current value of bp and its old value (old_bp) captured in the last CSSC. This information is sent to memory servers (❷ → ⑩) before they reclaim regions. Before evacuation happens, each memory server must ensure that regions to be evacuated have all their pages evicted, to avoid inconsistency. To this end, the CPU server evicts all pages for each selected region (❶).

Task 2: Memory servers reclaim selected regions by moving out their live objects (⑩ – ⑭). For these regions, their tracing (*i.e.*, closure computation) is already performed during the MSCT, and hence, reclamation simply follows the closure to copy out live objects (*i.e.*, object evacuation) from old regions into new ones. Object evacuation is done using a region's forward table, which is computed in traversal order to improve locality, as discussed earlier in §4.2. Live objects from multiple old regions can be compacted into a new region to reduce fragmentation. Moreover, each memory server attempts to coalesce regions connected by pointers, again, to improve locality — if region A has references from region B, *Semeru* attempts to copy live objects from A and B into the same (new) region. The new addresses of these objects can be computed easily by adding their offsets from the forward tables onto the base addresses of their target spaces (which may be brand-new or half-filled regions).

Since objects are moved, their addresses have changed and hence pointers (stack variables or fields of other objects) referencing the objects must be updated. Pointer updates, however, must be done through the CPU server, because pointers can be scattered across the cache and other memory servers. Thus after reclaiming regions, each memory server

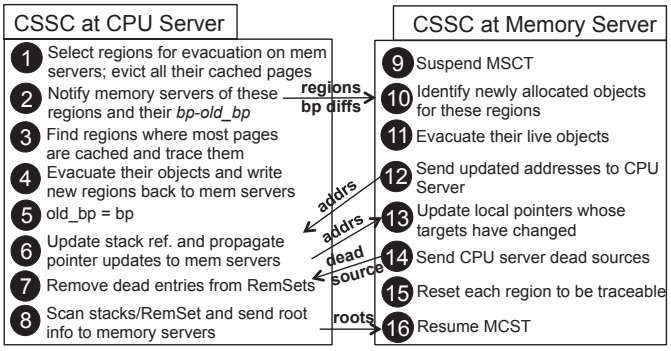


Figure 7: *Semeru*'s CSSC protocol: edges represent communications in the RDMA control path; $bp - old_bp$ represents the difference between the current bp and the value of bp captured at the last CSSC.

sends the updated addresses of moved objects back to the CPU server (⓬). If a cached object references a moved object, the CPU server updates the pointer directly; the CPU server must also propagate these update requests to other memory servers (❻ → ⓭), which may host objects referencing moved objects.

Task 3: While memory servers reclaim their regions, the CPU server reclaims regions where most objects are cached. Since these regions have not been traced during the MSCT, the CPU server has to trace them to build the closure and then reclaim them using the same object evacuation algorithm (❸ and ❹). Unlike memory-server region reclamation, the CPU server has to additionally write new regions back to their respective memory servers after object evacuation to ensure consistency (❹). Next, the CPU server remembers the current value of bp into old_bp (❺) for use in the next CSSC.

Task 4: Since most dead objects have already been reclaimed, the CPU server scans the remsets to remove *dead entries* (❼). This is important since otherwise remsets can keep growing and dead entries would become memory leaks. Removing dead entries at the CPU server requires memory servers to provide information about which objects are dead (⓮ → ❼) because most regions are traced and reclaimed at memory servers. The CPU server then scans each reference in each region's remset and removes those references with dead targets. Finally, the CPU server scans its stacks and the updated remset of each region to compute new roots, which are sent to memory servers for the next round of MSCT (❽). Memory servers reset the metadata (*e.g.*, `live_map` and forward table) so that the next round of MSCT can trace each region from scratch (⓯ and ⓰).

Since each CSSC only collects selected regions, it may not reclaim enough memory for the application to run. In such rare cases (*e.g.*, one or two in our experiments with each Spark application), *Semeru* runs a full-heap scan (*i.e.*, the same as a regular full-heap GC in G1), which brings all objects into the cache for tracing and collection. Since CSSC relies on remset-based modular tracing, it cannot reclaim dead objects that are (1) in different regions and (2) form cycles. Such objects have to be reclaimed at a full-heap GC.

5 The *Semeru* Swap System

We build *Semeru*'s swap system by piggybacking on Mellanox's NVMe-oF implementation [1]. This section briefly describes our modifications. During booting, the CPU server sends JVM metadata (such as metadata of loaded classes) in its *native heap* to memory servers, which use such information to launch LJVMs. On each memory server, the LJVM receives these native objects and reconstructs their virtual tables for function calls to execute correctly on these objects.

Block Layer. We modify NVMe-oF's block layer to add support for remote memory management. The remote physical memory that backs the Java heap on all memory servers is registered as a whole as an RDMA buffer and pinned down throughout the execution. As a simple optimization, we remove block-layer staging queues and merge several block I/O (BIO) requests into a single I/O request, turning them directly into RDMA messages.

Merging BIOs enables the use of InfiniBand's gather-scatter for data transfer. For each BIO request generated by the block layer, it often contains multiple physical pages to be transferred to a memory server. These physical pages are not necessarily contiguous. One optimization here is instead of generating multiple RDMA messages separately for these physical pages, we amortize per-message overhead by leveraging the scatter-gather technique so that these pages can be processed using a single RDMA message. We also develop thread-local RDMA message pools so that multiple threads can perform their own RDMA message creation and initialization without needing synchronization.

RDMA Management. All communications between the CPU and memory servers are through reliable one-sided RDMA. We distinguish these communications based on data types: (1) page fetching and evictions, which dominate the communications, go through a *data path* inside the kernel (to provide transparency to applications); (2) signals and GC information (*e.g.*, all messages in Figure 7), are passed through a *control path* implemented as a user-space library for efficiency. A user-space implementation benefits from efficiency from raw RDMA (*e.g.*, no overhead from system calls); since the control path does not overlap with the data path and transfers small amounts of information (*i.e.*, only inside each CSSC), our implementation can deliver good performance for both control and data paths.

6 Evaluation

To implement *Semeru*, we wrote/modified 58,464 lines of (non-comment) C/C++ code, including 43,838 lines for the LJVM (based upon OpenJDK version 12.0.2) on memory servers, 7,406 lines for the CPU-server JVM, and 7,220 lines for the Linux kernel (4.11-rc8). Our kernel support contains 4,424 lines of C code for the paging system and RDMA

management (based upon NVMe-oF), and 2,796 lines for the modified block layer and memory management part as well as new system calls.

Setup and Methodology. We ran *Semeru* in a cluster with one CPU server and three memory servers. Each server has two Xeon(R) CPU E5-2640 v3 processors, 128GB memory, one 200GB SSD, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter. Servers are connected by one Mellanox 100 Gbps InfiniBand switch. To emulate the weak compute of memory servers, we let the LJVM on each memory server use only one core. All our experiments used a 32GB heap, 512MB regions, and 4K pages. The default swap prefetching mechanism in Linux was used.

Unfortunately, we were only able to gain exclusive use of a small cluster with four machines when evaluating *Semeru*. Despite running on this small cluster, our experiments used large-scale applications involving multiple memory servers, representing a real-world use of *Semeru*. Adding more memory servers would not change the results because (1) memory servers perform modular collection — they do not communicate with each other and hence not have scalability issues; and (2) the CPU server only communicates with memory servers during each CSSC — more memory servers would only increase the control-path communication, which is minimal. Adding CPU servers and running more processes would increase the amount of tracing work on each memory server. However, as shown in §6.3, tracing for a large Spark application can only utilize 13% of each memory server's compute — one single core on each server can support simultaneous tracing for ∼8 Spark applications.

Name	Dataset	Size
GraphX-ConnectedComponents (**GCC**) GraphX-PageRank (**GPR**)	Wikipedia English [5]	2GB
Naïve-PageRank (**NPR**)	Wikipedia Polish [5]	1GB
Naïve TriangleCounting (**NTC**)	Synthetic 2.5K points 10K edges	1GB
MLlib-Bayes Classifiers (**MBC**)	KDD 2012 [4]	5GB

Table 1: Description of five Spark programs.

Name	Dataset	Size
Word Count (**FWC**)	Wikipedia English [5]	2GB
KMeans (**KMS**)		
Connected Components (**FCC**)		

Table 2: Description of three Flink batch-processing programs.

We evaluated *Semeru* with two widely deployed data analytics systems: Apache Spark (3.0.0) and Apache Flink (1.10.1). Spark was executed under Hadoop 3.2.1 and Scala 2.12.11, using a set of five programs (listed in Table 1): PageRank (GPR) and ConnectedComponents (GCC) from the GraphX [48] libraries, as well as Bayes Classifier (MBC) from the MLlib libraries. We also included naïve PageRank (NPR) and naïve TriangleCounting (NTC), implemented directly atop Spark.

Flink also ran on top of Hadoop version 3.2.1. Flink has both streaming and batch-processing models. In this experiment, we focused on the batch-processing model, in particular, Map/Reduce programs. The programs and their datasets are summarized in Table 2. These programs are selected based on their popularity and usefulness, covering a spectrum of text analytics, graph analytics, and machine learning tasks.

6.1 Overall *Semeru* Performance

We compared *Semeru* and the original OpenJDK 12 that runs the G1 GC — the default GC in the JVM since OpenJDK 9. G1 is a concurrent GC that runs concurrent tracing as the mutator thread executes and stops the world for memory reclamation. G1 is designed for short latency (*i.e.*, GC pauses) at the cost of reduced throughput (*i.e.*, concurrent tracing slows down the mutator as it competes resources with the mutator). We have tested other GCs as well and found that G1 consistently outperforms all others in latency.

We ran G1 with two swap mechanisms: a local RAMDisk and NVMe-oF, which connects the CPU server to remote memory on the three memory servers. To use NVMe-oF, we configured remote memory as remote RAMDisks, which host data objects without supporting memory management. *Semeru* ran on our own swap system built on top of NVMe-oF with added support for the remote heap and memory management. Each memory server hosts around one-third of the 32GB Java heap. There are three cache configurations: 100%, 50%, and 25%. The 100% configuration is our baseline, which represents the original OpenJDK's performance without any swapping.

Running Time. Figure 8 shows performance comparison between these systems, for our eight programs, under the three cache configurations. There is only one bar under the 100% cache configuration, representing the original performance of G1 that does not perform swapping.

	50% Cache			25% Cache		
System	**Mutator**	**GC**	**All**	**Mutator**	**GC**	**All**
G1-RD	1.82×	2.79×	1.87×	3.16×	4.59×	3.23×
G1-NVMe	2.00×	4.44×	2.24×	3.85×	14.13×	4.58×
Semeru	**1.06**×	**1.42**×	**1.08**×	**1.22**×	**2.67**×	**1.32**×

Table 3: Overhead summary: overheads are calculated using the G1 performance under the 100% cache configuration as the baseline.

Table 3 summarizes the time overheads incurred by memory disaggregation on these systems. The baseline used to calculate these overheads is the G1 performance under the 100% cache ratio (without any kernel and JVM modification). On average, G1 has 1.87× and 2.24× end-to-end overhead under RAMDisk and NVMe-oF, respectively, for the 50% cache configuration. When the cache ratio reduces to 25%, these overheads increase to 3.23× and 4.58×, respectively. By offloading tracing and evacuation to memory servers and improving the locality for the mutator threads, *Semeru* reduces these overheads, by **3.23 times overall**, to **1.08**× and **1.32**× for the two cache ratios, respectively.

Our first observation here is that disaggregation incurs a much higher overhead on GC than the mutator for Spark

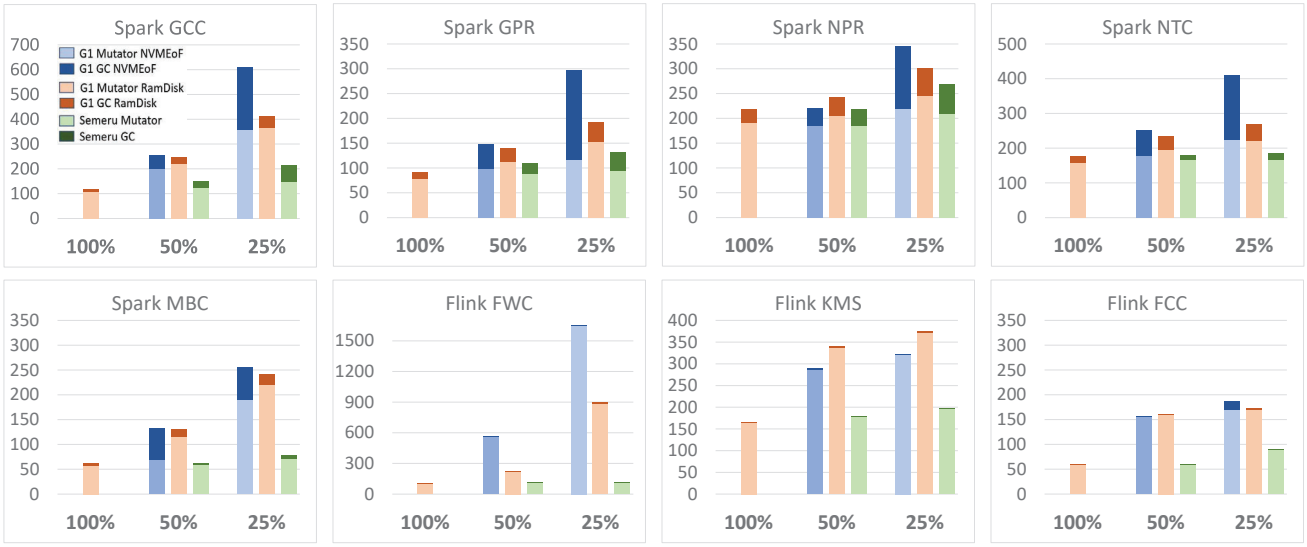


Figure 8: Performance comparisons between G1 under NVMe-oF (left bar of each group), G1 under RAMDisk (middle bar), and Semeru (right bar) for three cache configurations: 100%, 50%, and 25%; each bar is broken down into mutator (bottom) and GC (top) time (second).

applications, and it is consistent with our motivating data reported in §2. This is because GC algorithms inherently do not possess good locality and, as a result, pay a higher penalty for remote memory fetching than the mutator. This overhead grows significantly when the cache size decreases. It is also easy to see that accessing remote memory (via NVMe-oF) incurs a higher overhead than accessing the local RAMDisk.

The second observation is that for Flink, which has much less GC than Spark, *Semeru* can still considerably improve its performance. An inspection found that Flink stores data in the serialized form and implements operators that can process data without creating objects for them. Flink allocates long-lived data items directly in native memory and/or reserved space in the old generation. Nevertheless, *Semeru*'s optimizations are still effective. This is because the G1 GC uses a disaggregation-agnostic policy to dynamically tune the size of young generation. Since most objects in Flink die in the young generation, the pause time of each young GC is extremely short (*e.g.*, less than 10 ms) and always meets G1's pause-time target. As such, G1 keeps increasing the young generation size to reduce the GC frequency, making the young generation quickly reach the size of the CPU cache.

However, the problem here is the young generation contains large amounts of garbage, cached on the CPU server, leaving little cache space for long-lived data. This causes hot, long-lived data (*e.g.*, in native memory) to be frequently swapped in and out. In contrast, under *Semeru*'s region design, a CSSC is triggered when *Semeru* observes frequent swapping. The CSSC reclaims garbage and compacts regions, freeing up cache space for accommodating other hot data.

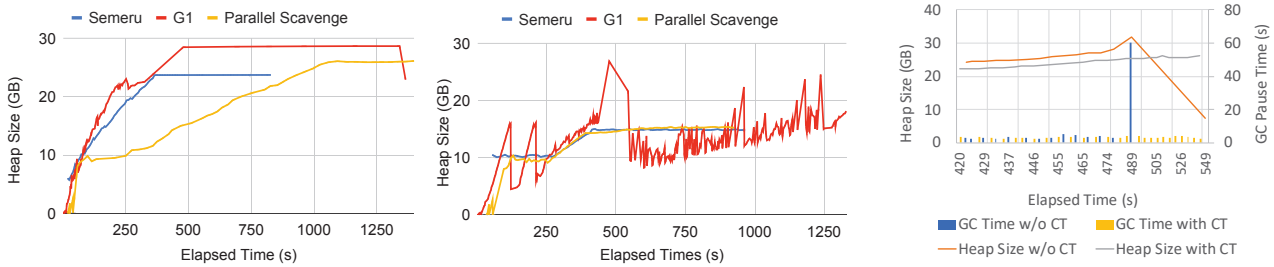
The third observation is that applications have different levels of tolerance to fetching latency. For example, GCC and GPR have an exceedingly high GC overhead because they create large RDDs and *persist* them in memory. These RDDs and their elements quickly become old and get promoted to the *old generation*. G1 cannot reclaim much memory in nursery GCs and, as such, most GCs scan the entire heap, requiring many remote fetches. For other applications such as Spark NPR, their GC performance is not as significantly degraded because their executions generate many temporary objects that die young (rather than old objects) — when a nursery GC runs, most young objects are garbage cached locally on the CPU server, and hence, they can be easily reclaimed without triggering many remote fetches.

System	50% Cache			25% Cache		
	Mutator	**GC**	**All**	**Mutator**	**GC**	**All**
G1-RD	1.73×	2.31×	1.75×	2.65×	2.35×	2.56×
G1-NVMe	1.91×	4.20×	2.10×	3.31×	5.61×	3.69×

Table 4: Summary of performance improvements achieved by *Semeru*: improvements are computed with $\frac{a}{b}$ where a is the (mutator, GC, or end-to-end) time under a system and cache configuration, and b is *Semeru*'s time under the same configuration.

To make *Semeru*'s improvements clear, Table 4 reports detailed improvement ratios under each configuration. It is easy to see that *Semeru* improves the performance of both the mutator and GC. On the mutator side, *Semeru* eliminates G1's concurrent marking — which runs on the CPU server in parallel with application execution, competing for resources with mutator threads and polluting the cache — and dynamically improves locality (discussed in §4.3) by relocating objects likely to be accessed to contiguous memory. On the GC side, *Semeru* significantly reduces pause time by letting memory servers perform tracing and evacuation, all of which used to be done on the CPU server.

Memory. To understand *Semeru*'s ability to reclaim memory, we collected post-GC memory footprints for Spark NPR and Spark KMS under three GCs: *Semeru*, G1, and Parallel



(a) Memory: NPR, 50% cache (b) Memory: KMS, 25% cache (c) Mem/time: NPR, w/ and w/o cont. tracing

Figure 9: Memory footprints under *Semeru*, G1, and Parallel Scavenge for NPR (a) and KMS (b); (c) shows the memory footprint and GC pause time with and without continuous tracing for NPR.

Scavenge (PS). PS is a non-concurrent GC designed for high throughput. We added PS because it often can reclaim more memory at each GC than G1 at the cost of higher latency. PS's strong memory reclamation capabilities are clearly seen in Figure 9(a) where PS has the lowest memory footprint throughout the execution. *Semeru* outperforms G1 — G1 uses concurrent tracing to estimate a garbage ratio for each region; with this information, when each STW phase runs, the GC can selectively reclaim regions with the highest garbage ratios. Under memory disaggregation, however, concurrent tracing runs slowly due to frequent remote fetches. It cannot finish tracing the heap at the time a STW starts; as a result, garbage ratios are not available for most regions.

As a result, at each STW phase, there is not much information about which regions have the most garbage, and thus, the GC selects arbitrary regions to collect. Many such regions do not have much garbage, which explains why G1 reclaims less memory than *Semeru* and PS. Note that *Semeru* does not suffer from this problem because tracing is done locally on memory servers; hence, it runs efficiently and can trace many regions between two consecutive CSSCs.

Figure 9(b) shows the memory footprint for Spark KMS running under the 25% cache configuration. In this case, *Semeru*'s collection performance is close to that of PS — for both of them, the program's memory consumption becomes stabilized after about 400 seconds. Under G1, however, the memory footprint fluctuates, again due to the (semi-random) selection of regions to collect. If regions with large garbage ratios happen to be in the cache, G1 is able to quickly identify them during concurrent marking and collect them in a subsequent STW phase. However, if they are remotely resident on memory servers, G1 would lack sufficient information in a STW phase to collect the right regions.

6.2 Effectiveness of Continuous Tracing

To understand the usefulness of continuous tracing on memory servers, we compared *Semeru* with a variant that does not perform continuous tracing but rather traces regions in each CSSC. In this variant, tracing is still done on memory servers but combined with other memory management tasks such as object evacuation in each STW phase. Without continuous tracing, which uses idle resources on memory servers to trace local regions, *Semeru* suffers from the same problem as G1 — when a CSSC runs, *Semeru* does not know which regions have the most garbage and thus should be reclamation targets. To minimize the GC latency, each CSSC has to be extremely short, leaving memory servers insufficient time to trace many regions. As a result, memory servers can only trace and reclaim regions based on their age without the more useful information of their garbage ratio.

To illustrate this problem, Figure 9(c) shows the post-GC memory footprint (*i.e.*, y-axis on the left) and the pause time of each CSSC (*i.e.*, y-axis on the right). The two lines represent the memory footprints of *Semeru* with and without continuous tracing while the short bars report the GC pauses. We make two important observations here. First, *Semeru* with continuous tracing consistently reclaims more memory than the version without continuous tracing, because it knows the right regions to reclaim in each CSSC. Second, since the version without continuous tracing cannot reclaim enough memory, it triggers a full-heap scan at the 484th second, which is extremely time consuming (*i.e.*, 65 seconds).

A modern generational GC achieves its efficiency by scanning only the young nursery generation in most of its GC runs. As soon as it needs to scan the entire heap, its performance degrades significantly. This is especially the case with memory disaggregation where a full-heap scan fetches most objects from memory servers to the CPU server, incurring an extremely long pause, as shown in the figure. The full-heap GC reclaims much space and reduces memory consumption.

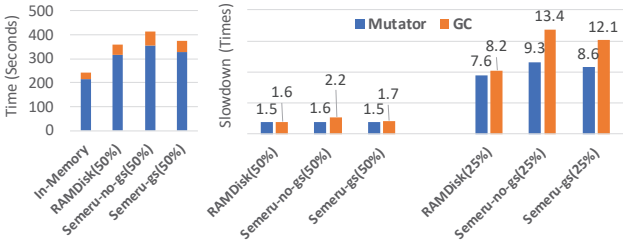
In contrast, with continuous tracing, *Semeru* does not encounter any full-heap GC throughout the execution. Although it does not reclaim as much memory as a full-heap GC, it avoids long pauses and yet is still able to give the application enough memory to run.

6.3 Tracing Performance

Memory servers are expected to possess weak compute power. To understand how tracing performs under different levels of compute, we used one single core on each memory and varied its frequency with DVFS. Table 5 summarizes the impact of each frequency on the tracing performance, GC and mutator performance, and end-to-end performance of NPR. We also obtained the same measurements when tracing is performed on the CPU server with a dedicated core. As shown, even with a single core at 1.2GHz, tracing on memory servers still yields

Configuration	Tracing Performance				Overall Performance		
	Thruput	CUtil	AT	AIT	GC	Mutator	Overall
(Memory Server) single core, 1.2 GHz	418.3 MB/s	29.0%	6.5 secs	4.6 secs	59.4 secs	180.2 secs	239.6 secs
(Memory Server) single core, 2.6 GHz	922.2 MB/s	12.4%	5.7 secs	5.0 secs	59.3 secs	173.9 secs	233.2 secs
(CPU Server) single core, 2.6 GHZ, dedicated to GC	93.9 MB/s	N/A	38.8 secs	N/A	126.0 secs	218.9 secs	344.9 secs

Table 5: Performance of NPR when tracing is performed under different core frequencies at memory servers: reported are the configurations (**Configuration**) of memory-server cores, tracing throughput (**Thruput**), memory-server CPU utilization (**CUtil**), average time between two consecutive CSSCs (**AT**), average idle CPU time between two consecutive CSSCs (**AIT**), total GC (**GC**) and mutator time (**Mutator**), and end-to-end run time (**Overall**).



(a) (b)

Figure 10: Comparisons between *Semeru*'s swap system and local RAMDisk: (a) shows Spark running times when the size of the cache is 50% of the heap size; the first bar reports performance of the baseline (cache ratio = 100%); (b) shows normalized performance (*i.e.*, slowdowns) for the two cache configurations (50% and 25%).

a throughput **4.5**× higher than doing so on the CPU server with a dedicated 2.6GHz core. This is easy to understand: the bottleneck of a memory-disaggregated system is at (1) the poor locality, which triggers many on-demand swaps, and (2) racing for network resource between the mutator and GC threads, *not* the lack of compute power.

Another important observation is on the low CPU utilization on memory servers. Even with a 1.2GHz core, continuous tracing between consecutive CSSCs has only 29% CPU utilization — this is because (1) tracing only follows pointers, (2) dead objects are not traced and hence, for each region, only a small fraction needs to be traced, and (3) not all regions need to be traced (*i.e.*, those with a high rate of cached objects are not traced). These results demonstrate that supporting multiple processes, with weak compute on memory servers, should not be a concern.

6.4 Swap Performance

To evaluate our swap system's performance, we turned off the *Semeru* runtime (*i.e.*, all memory management tasks on memory servers) and ran the original G1 GC on top of our swap system. We tried to run InfiniSwap [49], but its executions were frequently stuck, even on native programs. This subsection focuses on comparisons of swap performance between local RAMDisk and *Semeru*'s swap system (with and without using InfiniSwap's gather/scatter).

The results of Spark NPR are reported in Figure 10. We used two cache configurations: 50% and 25%. Figure 10(a) shows actual running times when the cache ratio is 50% between four versions of the system: in-memory (*i.e.*, cache ratio is 100%), RAMDisk, *Semeru*-no-gs (*i.e.*, gather/scatter is not used), and *Semeru*-gs (which uses gather/scatter). For ease of comparison, Figure 10(b) shows normalized times.

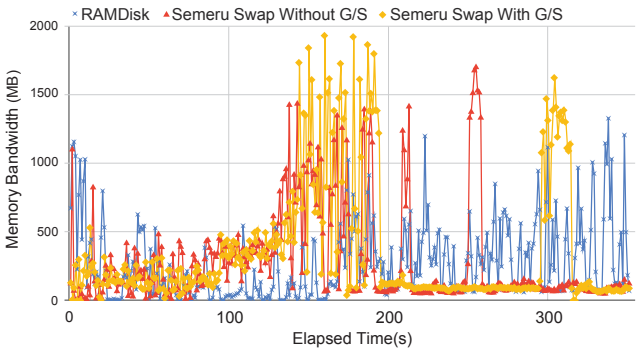


Figure 11: A comparison of the combined swap read/write throughput between *Semeru*-gs, *Semeru*-no-gs, and RAMDisk.

Under the 50% cache configuration, using RAMDisk as the swap partition incurs a 1.5× and 1.6× overhead in the mutator and GC, respectively, compared with the in-memory baseline. *Semeru*-no-gs increases the overheads to 1.6× and 2.2×. Merging BIO requests and using gather/scatter brings the overheads down to 1.5× and 1.7×, which are on par with those of the RAMDisk. Similar observations can be made for the 25% cache rate. Across all programs, gather/scatter improves the swap performance overall by 14%.

Figure 11 compares the read/write throughput between *Semeru*-gs, *Semeru*-no-gs, and RAMDisk when Spark LRG is executed under the 25% cache configuration. As shown, gather/scatter helps *Semeru* achieve a higher peak read/write bandwidth than *Semeru*-no-gs (especially when pages contiguously swapped come from / go to the same region).

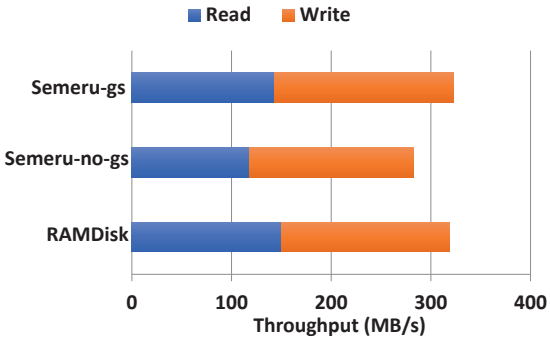


Figure 12: Average read/write throughput.

A comparison on the average read/write throughput between the three systems is shown in Figure 12. *Semeru*-gs's

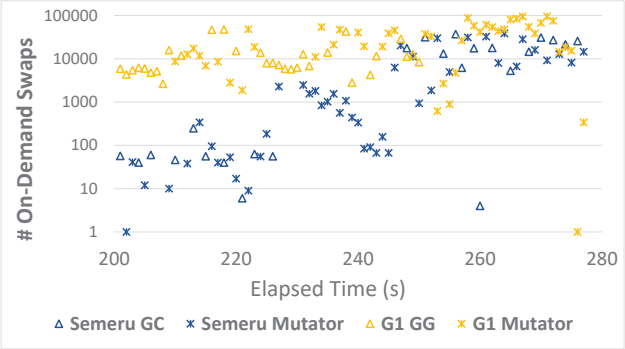


Figure 13: Numbers of on-demand swap-ins between G1 and *Semeru* under the 25% cache configuration for Spark MBC.

overall read/write throughput is 13% higher than that of *Semeru*-no-gs and is on par with that of RAMDisk. Clearly, additional gains can be obtained by merging BIO requests and using gather-scatter.

6.5 Locality Improvement

To understand how *Semeru* improves locality for application execution, we measured the number of on-demand swap-ins performed by the swap system under G1 and *Semeru* when Spark MBC was executed with a 25% cache ratio. Figure 13 reports how such numbers change as the execution progresses for both the mutator and GC. Both *Semeru*-muator and *Semeru*-GC need significantly fewer on-demand swap-ins due to improved locality. On average, *Semeru* reduces the number of on-demand swap-ins by **8.76**×. Note that both G1 and *Semeru* ran under the default swap prefetcher in Linux, which relies on the pages swapped in during the last two page faults: if they are contiguous, Linux continues to bring in several contiguous pages into the page cache; otherwise, it assumes that there are no patterns and reduces or stops prefetching. Despite the recent development of more advanced prefetchers (such as Leap [71]) for remote memory, *Semeru* already performs well under the default prefetcher in Linux. We expect it to continue to work well when other prefetchers are used. The average ratio between the sizes of data swapped in the data and control path is 29.8 across the programs.

7 Related Work

Resource Disaggregation. Due to rapid technological advances in network controllers, it has become practical to reorganize resources into disaggregated clusters [21, 29, 45, 51]. A disaggregated cluster can increase the hardware resource utilization and has the potential to overcome fundamental hardware limits, such as the critical "memory capacity wall" [9, 13, 17, 58, 67, 68, 95]. A good number of systems have been developed in the past to take advantage of this architecture [7, 35, 41, 42, 44, 54, 62]. However, almost all of them treat remote memory as fast storage. When the network connection only has microseconds of latency and hundreds of gigabits of bandwidth [55, 72], applications can suffer from significant delays in memory access. Despite many optimizations [7, 11, 49, 84, 87–89] developed to reduce this latency, they all focus on low-level system stacks and do not consider run-time characteristics of programs. They do not work well for managed cloud applications such as [6, 14, 15, 24–26, 31, 32, 50, 56, 75, 76, 81, 82, 92, 102–104, 106]. *Semeru* co-optimizes the runtime and the swap system, unlocking opportunities unseen by existing techniques.

Garbage Collection for Modern Systems. GC is a decades-old topic. In order to meet the requirements of low latency and high throughput, many concurrent GC algorithms have been proposed, including the Garbage-First (G1) GC [36], Compressor [59], ZGC [2], the Shenandoah GC [43], Azul's pauseless GC [34], and C4 [90], as well as several real-time GCs [18, 19]. These GC algorithms can run in the background with short pauses for mutator threads. However, none of them can work directly in the resource-disaggregated environment, which has a unique resource profile — data are all located on memory servers, the CPU server has a small cache, and memory servers have weak compute.

Efficiently using memory is important especially for applications running on the cloud [40]. Yak [79] is a region-based GC developed for such applications. Taurus [70] coordinates GC efforts in a distributed setting for cloud systems. Facade [80] uses region-based memory management to reduce GC costs for Big Data applications. Gerenuk [78] develops a compiler analysis and runtime system that enable native representation of data for managed analytics systems such as Spark and Hadoop. Espresso [99] and Panthera [95] are designed for systems with non-volatile memory. Platinum [98] is a GC that aims to reduce tail latency for interactive applications. NUMAGiC [47] is a GC developed for the NUMA architecture. However, NUMAGiC assumes that NUMA nodes are completely symmetric (with the same CPU, the same amount of local memory, and the same GC algorithm) — which is not the case for disaggregated clusters. DMOS [53] is a distributed GC algorithm that has not been implemented and whose performance in a real-world setting is unclear.

8 Conclusions

Semeru is a managed runtime designed for efficiently running managed applications with disaggregated memory. It achieves superior efficiency via a co-design of the runtime and swap system as well as careful coordination of different GC tasks.

Acknowledgments

We thank the OSDI reviewers for their valuable and thorough comments. We are grateful to our shepherd Yiying Zhang for her feedback, helping us improve the paper substantially. This work is supported by NSF grants CCF-1253703, CCF-1629126, CNS-1703598, CCF-1723773, CNS-1763172, CCF-1764077, CNS-1907352, CNS-1901510, CNS-1943621, CNS-2007737, CNS-2006437, and ONR grants N00014-16-1-2913 and N00014-18-1-2037, and a grant from the Alexander von Humboldt Foundation.

A Artifact Appendix

A.1 Artifact Summary

Semeru is a managed runtime built for a memory-disaggregated cluster where each managed application uses one CPU server and multiple memory servers. When launched on *Semeru*, the process runs its application code (mutator) on the CPU server, and the garbage collector on both the CPU server and memory servers in a coordinated manner. Due to task offloading and moving computation close to data, *Semeru* significantly improves the locality for both the mutator and GC and, hence, the end-to-end performance of the application.

A.2 Artifact Check-list

- **Hardware: Intel servers with InfiniBand**
- **Run-time environment: OpenJDK 12.02, Linux-4.11-rc8, CentOS 7.5(7.6) with MLNX-OFED 4.3(4.5)**
- **Public link: `https://github.com/uclasystem/Semeru`**
- **Code licenses: The GNU General Public License (GPL)**

A.3 Description

A.3.1 *Semeru*'s Codebase

Semeru contains the following three components:

- the Linux kernel, which includes a modified swap system, block layer and a RDMA module,
- the CPU-server Java Virtual Machine (JVM),
- the Memory-server lightweight Java Virtual Machine (LJVM).

These three components and their relationships are illustrated in Figure 14.

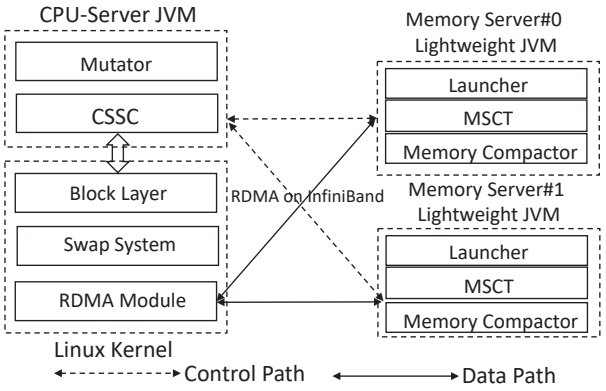


Figure 14: Overview of *Semeru*'s codebase.

A.3.2 Deploying *Semeru*

To build *Semeru*, the first step is to download its source code:

```
git clone
git@github.com:uclasystem/Semeru.git
```

When deploying Semeru, install the three components in the following order: the kernel on the CPU server, the *Semeru* JVM on the CPU server, and the LJVM on each memory server. Finally, connect the CPU server with memory servers before running applications.

Kernel Installation. We first discuss how to build and install the kernel.

- Modify `grub` and set `transparent_hugepage` to `madvise`:

```
sudo vim /etc/default/grub
+ transparent_hugepage=madvise
```

- Install the kernel and restart the machine:

```
cd Semeru/Linux-4.11-rc8
sudo ./build_kernel.sh build
sudo ./build_kernel.sh install
```

- Build the *Semeru* RDMA module:

```
# Add the IP of each memory server into
# Semeru/linux-4.11-rc8/include/
# linux/swap_global_struct.h
# e.g., the Infiniband IPs of the 2 memory servers
# are 10.0.0.2 and 10.0.0.4.
char* mem_server_ip[][] = {"10.0.0.2",
"10.0.0.4"};
uint16_t mem_server_port = 9400;
# Then build the Semeru RDMA module
make
```

Install the CPU-Server JVM. We next discuss the steps to build and install the CPU-server JVM.

- Download Oracle JDK 12 to build Semeru JVM:

```
# Assume jdk 12.02 is under path
# ${home_dir}/jdk12.0.2
# Or change the path in shell script
# Semeru/CPU-Server/build_cpu_server.sh
boot_jdk="${home_dir}/jdk12.0.2"
```

- Build the CPU-server JVM:

```
# ${build_mode} can be one of the three modes:
# slowdebug, fastdebug, or release.
# We recommend fastdebug to debug the JVM code
# and release to test the performance.
# Please make sure both the CPU server and
# memory servers use the same build mode.
cd Semeru/CPU-Server/
./build_cpu_server.sh ${build_mode}
./build_cpu_server.sh build
# Take fastdebug mode as example — the compiled
# JVM will be in:
# Semeru/CPU-Server/build
# /linuxx86_64serverfastdebug/jdk
```

Install the Memory-Server LJVM. The next step is to install the LJVM on each memory server.

- Download OpenJDK 12 and build the LJVM:

```
# Assume OpenJDK12 is under the path
#${home_dir}/jdk-12.0.2
# Or you can change the path in the script
# Semeru/Memory-Server/build_mem_server.sh
boot_jdk="${home_dir}/jdk-12.0.2"
```

- Change the IP addresses:

```
# E.g., mem-server #0's IP is 10.0.0.2, ID is 0.
# Change the IP address and ID in file:
# Semeru/Memory-Server/src/hotspot/share/
# utilities/globalDefinitions.hpp
#@Mem-server #0
#define NUM_OF_MEMORY_SERVER 2
#define CUR_MEMORY_SERVER_ID 0
static const char cur_mem_server_ip[] =
"10.0.0.2";
static const char cur_mem_server_port[]
= "9400";
```

- Build and install the LJVM:

```
# Use the same ${build_mode} as the CPU-server
# JVM.
cd Semeru/CPU-Server/
./build_memory_server.sh ${build_mode}
./build_memory_server.sh build
./build_memory_server.sh install
# The compiled Java home will be installed under:
# {home_dir}/jdk12u-self-build/jvm/
# openjdk-12.0.2-internal
# Set JAVA_HOME to point to this folder.
```

A.3.3 Running Applications

To run applications, we first need to connect the CPU server with memory servers. Next, we mount the remote memory pools as a swap partition on the CPU server. When the application uses more memory than the limit set by `cgroup`, its data will be swapped out to the remote memory via RDMA.

- Launch memory servers:

```
# Use the shell script to run each memory server.
#${execution_mode} can be execution or gdb.
#@Each memory server
cd Semeru/ShellScrip
run_rmem_server_with_rdma_service.sh
Case1 ${execution_mode}
```

- Connect the CPU server with memory servers:

```
#@CPU server
cd Semeru/ShellScript/
install_semeru_module.sh semeru
# To close the swap partition, do the following:
#@CPU server
cd Semeru/ShellScript/
install_semeru_module.sh close_semeru
```

- Set a cache size limit for an application:

```
# E.g., Create a cgroup with 10GB memory limita-
tion.
#@CPU server
cd Semeru/ShellScript
cgroupv1_manage.sh create 10g
```

- Add a Spark executor into the created `cgroup`:

```
# Add a Spark worker into the cgroup, memctl.
# Its sub-process, executor, falls into the same cgroup.
# Modify the function start_instance under
# Spark/sbin/start-slave.sh
#@CPU server
cgexec -sticky -g memory:memctl
"${SPARK_HOME}/sbin" /sparkdaemon.sh
start $CLASS $WORKER_NUM -webui-port
"$WEBUI_PORT" $PORT_FLAG $PORT_NUM
$MASTER "$@"
```

- Launch a Spark application:

  Some *Semeru* JVM options need to be added for both CPU-server JVM and LVJMs. CPU-server JVM and memory server LJVMs should use the value for the same JVM option.

```
# E.g., under the configuration of 25% local memmory
# 512MB Java heap Region
#@CPU server
-XX:+SemeruEnableMemPool
-XX:EnableBitmap -XX:-UseCompressedOops
-Xnoclassgc -XX:G1HeapRegionSize=512M
-XX:MetaspaceSize=0x10000000
-XX:SemeruLocalCachePercent=25
#@Each memory server
# ${MemSize}: the memory size of current memory
server
# ${ConcThread}: the number of concurrent threads
-XX:SemeruEnableMemPool
-XX:-UseCompressedOops
-XX:SemeruMemPoolMaxSize=${MemSize}
-XX:SemeruMemPoolInitialSize=${MemSize}
-XX:SemeruConcGCThreads=${ConcThread}
```

  More details of *Semeru*'s installation and deployment can be found in *Semeru*'s code repository.

References

[1] NVMe over fabrics. http://community.mellanox.com/s/article/what-is-nvme-over-fabrics-x.

[2] The Z garbage collector. https://wiki.openjdk.java.net/display/zgc/Main.

[3] SeaMicro Technology Overview. https://data.tiger-optics.ru//download/seamicro/SM_TO02_v1.4.pdf, 2010.

[4] Libsvm data: Classification. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets, 2012.

[5] Wikipedia networks data. http://konect.uni-koblenz.de/networks/, 2020.

[6] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proceedings of VLDB Endow.*, 1(1):958–969, 2008.

[7] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: A simple abstraction for remote memory. In *USENIX ATC*, pages 775–787, 2018.

[8] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SoCC*, pages 121–127, 2017.

[9] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing far memory data structures: Think outside the box. In *HotOS*, pages 120–126, 2019.

[10] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, pages 336–348, 2015.

[11] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[12] Amazon. Amazon EC2 root device volume. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html#RootDeviceStorageConcepts, 2019.

[13] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. In *HotCloud*, 2020.

[14] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[15] Apache Flink. http://flink.apache.org/.

[16] K. Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. In *FAST*, 2014.

[17] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[18] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *EMSOFT*, pages 245–254, 2008.

[19] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL*, pages 285–298, 2003.

[20] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[21] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA*, 2011.

[22] M. N. Bojnordi and E. Ipek. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *ISCA*, pages 13–24, 2012.

[23] M. N. Bojnordi and E. Ipek. A programmable memory controller for the DDRx interfacing standards. *ACM Trans. Comput. Syst.*, 31(4):11:1–11:31, 2013.

[24] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[25] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ISMM*, pages 119–130, 2013.

[26] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[27] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ISCA*, pages 225–236, 2012.

[28] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.*, 11(12):1849–1862, 2018.

[29] A. Carbonari and I. Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets-XVI*, pages 164–170, 2017.

[30] CCIX. Cache coherent interconnect for accelerators. https://www.ccixconsortium.com/, 2018.

[31] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[32] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[33] I.-H. Chung, B. Abali, and P. Crumley. Towards a composable computer system. In *HPC Asia*, pages 137–147, 2018.

[34] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *VEE*, pages 46–56, 2005.

[35] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *OSDI*, 1994.

[36] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, pages 37–48, 2004.

[37] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[38] Facebook. Introducing Lightning: A flexible NVMe JBOF. https://code.fb.com/data-center-engineering/introducing-lightning-a-flexible-nvme-jbof, 2019.

[39] Facebook and Intel. Facebook and intel collaborate on future data center rack technologies. http://goo.gl/6h2Ut, 2013.

[40] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*, pages 394–409, 2015.

[41] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, pages 201–212, 1995.

[42] E. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. In *University of Washington CSE TR CSE TR*, 1991.

[43] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *PPPJ*, pages 13:1–13:9, 2016.

[44] M. D. Flouris and E. P. Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4), Dec 1999.

[45] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[46] GenZ. Genz consortium. http://genzconsortium.org/, 2019.

[47] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: A garbage collector for big data on big NUMA machines. In *ASPLOS*, pages 661–673, 2015.

[48] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[49] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[50] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI*, pages 121–133, 2012.

[51] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[52] Hewlett-Packard. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/.

[53] R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage collecting the world: One car at a time. In *OOPSLA*, pages 162–175, 1997.

[54] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. In *Digest of Papers. Compcon Spring*, pages 538–547, Feb 1993.

[55] Intel. Intel high performance computing fabrics. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/, 2019.

[56] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[57] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[58] K. Keeton. The Machine: An architecture for memory-centric computing. In *ROSS*, 2015.

[59] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental, and parallel compaction. In *PLDI*, pages 354–363, 2006.

[60] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash storage disaggregation. In *EuroSys*, pages 29:1–29:15, 2016.

[61] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, pages 345–359, 2017.

[62] S. Koussih, A. Acharya, and S. Setia. Dodo: a user-level system for exploiting idle memory in workstation clusters. In *HPDC*, pages 301–308, Aug 1999.

[63] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[64] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, pages 84–92, 1996.

[65] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. Understanding rack-scale disaggregated storage. In *HotStorage*, 2017.

[66] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[67] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, pages 267–278, 2009.

[68] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, pages 1–12, 2012.

[69] M. Maas, K. Asanović, and J. Kubiatowicz. A hardware accelerator for tracing garbage collection. In *ISCA*, pages 138–151, 2018.

[70] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS*, pages 457–471, 2016.

[71] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[72] Mellanox. Connectx-6 single/dual-port adapter supporting 200gb/s with vpi. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card, 2019.

[73] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *NSDI*, pages 257–273, 2014.

[74] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, pages 245–260, 2007.

[75] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *PLDI*, pages 121–131, 2011.

[76] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[77] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *NSDI*, pages 17–33, 2017.

[78] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu. Gerenuk: Thin computation over big native data using speculative program transformation. In *SOSP*, pages 538–553, 2019.

[79] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[80] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, pages 675–690, 2015.

[81] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX ATC*, pages 267–273, 2008.

[82] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[83] OpenCAPI. Open coherent accelerator processor interface. https://opencapi.org/, 2018.

[84] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[85] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, pages 13–24, 2014.

[86] S. M. Rumble. Infiniband verbs performance. https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance, 2010.

[87] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[88] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, pages 255–270, 2019.

[89] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart remote memory. In *EuroSys*, 2020.

[90] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In *ISMM*, pages 79–88, 2011.

[91] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *SOSP*, pages 306–324, 2017.

[92] Storm: dstributed and fault-tolerant realtime computation. https://github.com/nathanmarz/storm.

[93] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *PSDE*, pages 157–167, 1984.

[94] VMware. Virtual SAN. https://www.vmware.com/products/vsan.html, 2019.

[95] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *PLD*, pages 347–362, 2019.

[96] W.-H. Wang, J.-L. Baer, and H. M. Levy. Readings in computer architecture. chapter Organization and Performance of a Two-level Virtual-real Cache Hierarchy, pages 434–442. 2000.

[97] Wen-Hann Wang, J. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *ISCA*, pages 140–148, 1989.

[98] M. Wu, Z. Zhao, Y. Yang, H. Li, H. Chen, B. Zang, H. Guan, S. Li, C. Lu, and T. Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *USENIX ATC*, 2020.

[99] M. Wu, Z. Ziming, L. Haoyu, L. Heting, C. Haibo, Z. binyu, and G. Haibing. Espresso: Brewing Java for more non-volatility. In *ASPLOS*, pages 70–83, 2018.

[100] G. Xu. Finding reusable data structures. In *OOPSLA*, pages 1017–1034, 2012.

[101] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[102] G. H. Xu, M. Veanes, M. Veanes, M. Musuvathi, T. Mytkowicz, B. Zorn, H. He, and H. Lin. Niijima: Sound and automated computation consolidation for efficient multilingual data-parallel pipelines. In *SOSP*, pages 306–321, 2019.

[103] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[104] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[105] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for Azure. In *NSDI*, pages 519–532, 2018.

[106] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*, pages 1060–1071, 2010.



Caladan: Mitigating Interference at Microsecond Timescales

Joshua Fried, Zhenyuan Ruan, Amy Ousterhout†, Adam Belay
MIT CSAIL, †UC Berkeley

Abstract

The conventional wisdom is that CPU resources such as cores, caches, and memory bandwidth must be partitioned to achieve performance isolation between tasks. Both the widespread availability of cache partitioning in modern CPUs and the recommended practice of pinning latency-sensitive applications to dedicated cores attest to this belief.

In this paper, we show that resource partitioning is neither necessary nor sufficient. Many applications experience bursty request patterns or phased behavior, drastically changing the amount and type of resources they need. Unfortunately, partitioning-based systems fail to react quickly enough to keep up with these changes, resulting in extreme spikes in latency and lost opportunities to increase CPU utilization.

Caladan is a new CPU scheduler that can achieve significantly better quality of service (tail latency, throughput, etc.) through a collection of control signals and policies that rely on fast core allocation instead of resource partitioning. Caladan consists of a centralized scheduler core that actively manages resource contention in the memory hierarchy and between hyperthreads, and a kernel module that bypasses the standard Linux Kernel scheduler to support microsecond-scale monitoring and placement of tasks. When colocating memcached with a best-effort, garbage-collected workload, Caladan outperforms Parties, a state-of-the-art resource partitioning system, by 11,000×, reducing tail latency from 580 ms to 52 μs during shifts in resource usage while maintaining high CPU utilization.

1 Introduction

Interactive, data-intensive web services like web search, social networking, and online retail commonly distribute requests across thousands of servers. Minimizing tail latency is critical for these services because end-to-end response times are determined by the slowest individual response [4, 14]. Efforts to reduce tail latency, however, must be carefully balanced with the need to maximize datacenter efficiency; large-scale datacenter operators often pack several tasks together on the same machine to improve CPU utilization in the presence of variable load [22, 57, 66, 71]. Under these conditions, tasks must compete over shared resources such as cores, memory bandwidth, caches, and execution units. When shared resource contention is high, latency increases significantly; this slowdown of tasks due to resource contention is called *interference*.

The need to manage interference has led to the development of several hardware mechanisms that *partition* resources. For example, Intel's Cache Allocation Technology (CAT) uses way-based cache partitioning to reserve portions of the last level cache (LLC) for specific cores [21]. Many systems use these partitioning mechanisms to improve performance isolation [8, 12, 28, 38, 62, 73]. They either statically assign enough resources for peak load, leaving significant CPU utilization on the table, or else make dynamic adjustments over hundreds of milliseconds to seconds. Because each adjustment is incremental, converging to the right configuration after a change in resource usage can take dozens of seconds [8, 12, 38].

Unfortunately, real-world workloads experience changes in resource usage over much shorter timescales. For example, network traffic was observed to be very bursty in Google's datacenters, sometimes consuming more than a dozen cores over short time periods [42], and a study of Microsoft's Bing reports highly bursty thread wakeups on the order of microseconds [27]. Phased resource usage is also common. For example, we found that tasks that rely on garbage collection (GC) periodically consume all available memory bandwidth (§2). Detecting and reacting to such sudden changes in resource usage is not possible with existing systems.

Our goal is to maintain both high CPU utilization and strict performance isolation (for throughput and tail latency) under realistic conditions in which resource usage, and therefore interference, changes frequently. A key requirement is faster reaction times, as even microsecond delays can impact latency after an abrupt increase in interference (§2). There are two challenges toward achieving microsecond reaction times. First, there are many types of interference in a shared CPU (hyperthreading, memory bandwidth, LLC, etc.), and obtaining the right control signals that can accurately detect each of them over microsecond timescales is difficult. Second, existing systems face too much software overhead to either gather control signals or adjust resource allocations quickly.

To overcome these challenges, we present an interference-aware CPU scheduler, called **Caladan**. Caladan consists of a centralized, dedicated scheduler core that collects control signals and makes resource allocation decisions, and a Linux Kernel module, called KSCHED, that efficiently adjusts resource allocations. Our scheduler core distinguishes between high-priority, latency-critical (LC) tasks and low-priority, best-effort (BE) tasks. To avoid the reaction time limitations imposed by hardware partitioning (§3), Caladan relies exclusively on core allocation to manage interference.

Caladan uses a carefully selected set of control signals and corresponding actions to quickly and accurately detect and respond to interference over microsecond timescales. We

observe that interference has two interrelated effects: first, interference slows down the execution speed of cores (more cache misses, higher memory latency, etc.), impacting the *service times* of requests; second, as cores slow down, *compute capacity* drops; when it falls below offered load, queueing delays increase dramatically.

Caladan's scheduler targets these effects. It collects fine-grained measurements of memory bandwidth usage and request processing times, using these to detect memory bandwidth and hyperthreading interference, respectively. It then restricts cores from the antagonizing BE task(s), eliminating most of the impact on service times. For LLC interference, Caladan cannot eliminate service time overheads directly, but it can still prevent a decrease in compute capacity by allowing LC tasks to steal extra cores from BE tasks.

The KSCHED kernel module accelerates scheduling operations such as waking tasks and collecting interference metrics. It does so by amortizing the cost of sending interrupts, offloading scheduling work from the scheduler core to the tasks' cores, and providing a non-blocking API that allows the scheduler core to handle many inflight operations at once. These techniques eliminate scheduling bottlenecks, allowing Caladan to react quickly while scaling to many cores and tasks, even under heavy interference.

To the best of our knowledge, Caladan is the first system that can maintain both strict performance isolation and high CPU utilization under frequently changing interference and load. To achieve these benefits, Caladan imposes two new requirements on applications: the adoption of a custom runtime system for scheduling and the need for LC tasks to expose their internal concurrency (§8). In exchange, Caladan is able to converge to the right resource configuration 500,000× faster than the typical speed reported for Parties, a state-of-the-art resource partitioning system [12]. We show that this speedup yields an 11,000× reduction in tail latency when colocating memcached with a BE task that relies on garbage collection. Moreover, we show that Caladan is highly general, scaling to multiple tasks and maintaining the same benefits while colocating a diverse set of workloads (memcached, an in-memory database, a flash storage service, an x264 video encoder, a garbage collector, etc.). Caladan is available at https://github.com/shenango/caladan.

2 Motivation

In this section, we demonstrate how performance can degrade when interference is not quickly mitigated. Many workloads exhibit phased behavior, drastically changing the types and quantities of resources they use at sub-second timescales. Examples include compression, compilation, Spark compute jobs, and garbage collectors [49, 59]. The request rates issued to tasks can also change rapidly, with bursts occurring over microsecond timescales [5, 27, 42]; these bursts in load can cause bursts of resource usage. In both cases, abrupt changes

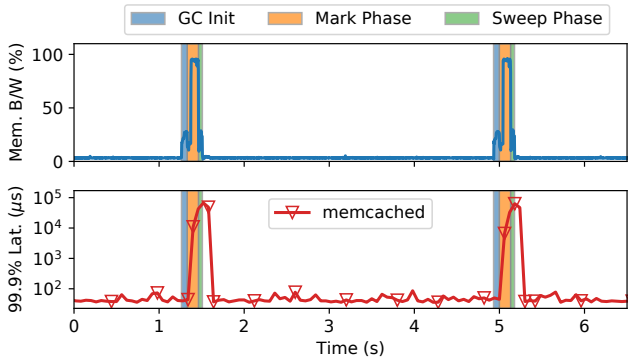


Figure 1: Periodic GC in a background task (shaded regions) increases memory bandwidth usage (top), causing severe latency spikes for a colocated memcached instance (bottom). Note the log-scaled y-axis in the bottom graph.

in resource usage can abruptly increase interference. This degrades request service times and causes request queues to grow when the rate of arriving requests exceeds the rate at which a task can process them.

To better understand the challenges associated with time-varying interference, we consider what happens when we colocate an LC task, memcached [43], with a BE workload that exhibits phased behavior due to garbage collection. In this example, we use the Boehm GC (see §7), which employs the mark-sweep algorithm to reclaim dead heap objects [10]. We have observed similar problems with more sophisticated, incremental GCs, such as the Go Language Runtime [60].

In this experiment, we offer a fixed load to memcached and statically partition cores between the two tasks. memcached is given enough cores to keep its 99.9$^{\text{th}}$ percentile tail latency below 50 μs when run in isolation. As shown in Figure 1, this allocation is sufficient to protect tail latency when the GC is not running but it fails when the GC starts. The GC pauses normal execution of the BE task for 100–200 ms and scans the entire heap using all cores available to the BE task, which saturates memory bandwidth. During this brief period, each memcached request experiences a higher rate of cache misses and larger memory access latencies, causing the rate at which memcached can service requests to drop by about half and queues to build up. As a result, memcached's queueing delay increases at a rate of 5 μs every 10 μs, eventually reaching a tail latency that is 1000× higher than normal.

This example illustrates that fixed core partitioning is insufficient, and also indicates what core reallocation speed is necessary in order to effectively mitigate interference. If changes in interference can instantaneously reduce the request service rate by half, then in order to keep latencies from increasing by X, the CPU scheduler must detect and respond to interference within $2X$. Thus, preventing a latency increase of 50 μs requires reaction times within 100 μs. Unfortunately, existing systems are not designed to respond this quickly (§3), forcing datacenter operators to either tolerate severe tail latency spikes, or else isolate these tasks on different servers.

System	Decision Interval	Typical Convergence	Requires CAT	Supports HT
Heracles [38]	2–15 s	30 s	✓	✗
Parties [12]	500 ms	10–20 s	✓	✗
Caladan	10–20 μs	20 μs	✗	✓

Table 1: A comparison of Caladan to state-of-the-art systems that use partitioning to manage interference. Caladan can converge to the right resource configuration 500,000× faster.

3 Background

Throughout this paper, we discuss three forms of interference that can occur when sharing a CPU: hyperthreading interference, memory bandwidth interference, and LLC interference. Hyperthreading interference is usually present at a baseline level whenever tasks are running on *sibling* cores because the CPU divides certain physical core resources (e.g., the micro-op queue), but it can become more severe depending on whether shared resources (L1/L2 caches, prefetchers, execution units, TLBs, etc.) are contended. Memory bandwidth and LLC interference, on the other hand, can vary in intensity, but impact all cores that share the same physical CPU. As memory bandwidth usage increases, memory access latency slowly increases due to interference, until memory bandwidth becomes saturated; access latency then increases exponentially [62]. LLC interference is determined by the amount of cache each application uses: when demand exceeds capacity, LLC miss rates increase.

In this section, we discuss why existing systems are unable to manage abrupt changes in interference (§3.1) and explore the limitations imposed on them by the hardware extensions available in commercial CPUs (§3.2).

3.1 Existing Approaches to Interference

State-of-the-art systems such as Heracles [38] and Parties [12] handle interference by dynamically partitioning resources, such as cores and LLC partition sizes. However, both Heracles and Parties make decisions and converge to new resource allocations too slowly to manage bursty interference (Table 1). There are two main reasons. First, both systems detect interference using application-level tail latency measurements, which must be measured over hundreds of milliseconds in order to obtain stable results; the Parties authors found that shorter intervals produced "noisy and unstable results" [12]. Second, both systems make incremental adjustments to resource allocations, gradually converging to a configuration that can meet latency objectives. These systems lack the ability to identify the source of interference (application and contended resource) directly, so convergence can involve significant trial-and-error as different resources are throttled, requiring seconds to converge to a new resource allocation. During the adjustment period, latency often continues to suffer because the LC task must wait to be given enough resources to reduce its queueing delay buildup.

Thus, both Heracles and Parties take at least 50× as long to adapt to changes in interference as the duration of a GC cycle in our example. As a result, operators must make tradeoffs based on tunable parameters: either tail latency tolerances (e.g., 99.9$^{\text{th}}$ percentile tail latency) can be set higher, causing the GC interference to be tolerated without resource reallocations, or they can be set lower, causing the GC workload to be throttled continuously. Because the GC workload causes minimal interference during the majority of its execution (while not collecting garbage), faster reaction times are needed to keep cores busy without compromising tail latency.

In addition to convergence speed, existing systems suffer from scalability limitations. For example, a typical datacenter server must handle several LC and BE tasks simultaneously [66, 71], but Heracles is limited to only a single LC task (and many BE tasks). Parties can support multiple LC and BE tasks, but because it can only guess at which task is causing interference, its convergence time increases with each additional task.

The hardware mechanisms on which these systems rely also impose limitations. For example, hyperthreads lack control over resource partitioning, so Heracles and Parties turn them off entirely. Using both hyperthreads on a core simultaneously can yield up to 30% higher throughput than using a single hyperthread [40, 44, 45, 54], so this lowers system throughput significantly. Furthermore, the available hardware partitioning mechanisms that can be controlled constrain both reaction speeds and scalability. We discuss this problem next.

3.2 Limitations of Hardware Extensions

Intel has added several extensions to its server CPUs that are designed to partition and monitor the LLC and memory bandwidth. These extensions are optimized for scenarios where resource demand changes slowly, but as shown in our study of the GC workload, this assumption does not always hold. To better understand these limitations, we discuss each component in more detail.

The most commonly used extension is CAT, a technology that divides portions of the LLC between tasks to increase performance determinism [21]. CAT's way-based hardware implementation suffers from two limitations. First, changes to the partition configuration can take considerable time to have an effect; Intel cautions that "a reduction in the performance of [CAT] may result if [tasks] are migrated frequently" [26, sec. 17.19.4.2]. Appropriately sizing a partition, however, is challenging under time-varying demand because it must be large enough to accommodate peak usage. Second, CAT must divide a finite number of set-associative ways between partitions, reducing associativity within each partition. Unfortunately, performance can degrade significantly as associativity decreases [56, 67]; KPart avoids this by grouping complementary tasks together in the same partition, but it relies on frequent online profiling to identify groupings, resulting in high tail latency [18].

Another extension called Memory Bandwidth Allocation (MBA) applies a per-core rate limiter to DRAM accesses to throttle bandwidth consumption. MBA is necessary for systems that statically assign cores because it is the only method they can use to limit bandwidth consumption. Unfortunately, it is at odds with our goal of achieving high CPU utilization: a core that is heavily rate-limited by MBA will spend the majority of its time stalling. Instead, we found it is more efficient to allocate fewer cores, achieving the same throughput for a task, but with higher per-core utilization.

Finally, configuring partitioning mechanisms effectively requires the attribution of resource usage to specific tasks. To help with this goal, Intel introduced Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM) [72]. Unfortunately, these mechanisms are unable to detect changes in system conditions quickly. For example, when monitoring a streaming task with CMT, it takes 112 ms for its cache occupancy measurement to stabilize [21]. Similarly, we discovered experimentally that MBM requires milliseconds to accurately estimate memory bandwidth usage.

4 Challenges and Approach

Our overarching goal is to maintain performance isolation while maximizing CPU utilization. Achieving this goal is difficult because managing changes in interference requires microsecond-scale reaction times. Partitioning resources in hardware is too slow for these timescales (§3.2), so Caladan's approach is to instead manage interference by controlling how cores are allocated to tasks. Prior systems have adjusted cores as part of their strategy for managing interference [12, 28, 38, 70], but Caladan is the first system to rely exclusively on core allocation to manage multiple forms of interference. To mitigate interference quickly enough, we had to overcome two key challenges:

1. **Sensitivity:** For fast and targeted reactions, Caladan requires control signals that can identify the presence of interference and its source—task and contended resource—within microseconds. Commonly used performance metrics like CPI [71] or tail latency [12, 38] (as well as hardware mechanisms like MBM and CMT) are too noisy to be useful over short timescales. Metrics like queueing delay [8, 42, 47, 68] can be measured over microsecond timescales, but cannot identify the source of interference, only that a task's performance is degrading.
2. **Scalability:** Existing systems depend heavily on the Linux Kernel in order to gather control signals and adjust resource allocations (e.g., using `sched_setaffinity()` to adjust core allocations) [8, 12, 20, 38, 47, 52, 68]. Unfortunately, Linux adds overhead to these operations, and these overheads increase in the presence of interference and as the number of cores and tasks increase.

We address the challenge of sensitivity by carefully selecting control signals that enable fast detection of interference and by dedicating a core to monitor these signals and take action to mitigate interference as it arises. We address the challenge of scalability with a Linux Kernel module named KSCHED. We describe these in more detail below.

Caladan's Actions to Mitigate Interference

	Impact of Interference	
Contended Resource	↑ Service Times	↓ Compute Capacity
Hyperthreads	idle sibling core	add victim cores
Memory Bandwidth	throttle antagonist	add victim cores
LLC	none	add victim cores

Table 2: When a resource (left) becomes contended, Caladan takes action to avoid increased service times (middle). When this is insufficient to maintain compute capacity, Caladan takes additional action (right).

4.1 Caladan's Approach

Caladan dedicates a single core, called the *scheduler*, to continuously poll and gather a set of *control signals* over microsecond timescales. The scheduler uses these signals to detect interference and then reacts by adjusting core allocations. The scheduler is designed to manage several forms of interference (§3), using control signals tailored to each. For hyperthreads, we assume interference is always present when both siblings are active (because some physical core resources are partitioned) and focus on reducing interference for the requests that will impact tail latency—that is, the longest running requests [70]. We measure request processing times to identify these requests. For memory bandwidth, we measure global memory bandwidth usage to detect DRAM saturation and measure per-core LLC miss rates to attribute usage to a specific task. For cases like the LLC where we cannot directly measure or infer interference, we can still measure a key side effect of interference: increased queueing delays, caused by reductions in compute capacity. By focusing on interference-driven control signals, Caladan can detect problems before quality of service is degraded.

Table 2 summarizes the actions Caladan takes to mitigate interference. We first try to prevent service time increases by reducing interference directly. For example, Caladan reduces hyperthreading interference by controlling which logical cores (hyperthreads) may be used, idling a logical core when its sibling exceeds a request processing time threshold. In addition, it reduces memory bandwidth interference by limiting how many cores each task may use; this is effective because reducing the number of cores allocated to a task reduces its memory bandwidth usage. However, reducing LLC interference is more difficult: the magnitude of LLC interference is determined primarily by how much LLC capacity a task uses, but reducing a task's number of cores reduces its LLC access rate rather than its LLC capacity. Therefore, Caladan compensates for LLC interference—and any remaining hyperthreading and memory bandwidth interference—by granting extra cores to victim tasks, allowing them to recoup

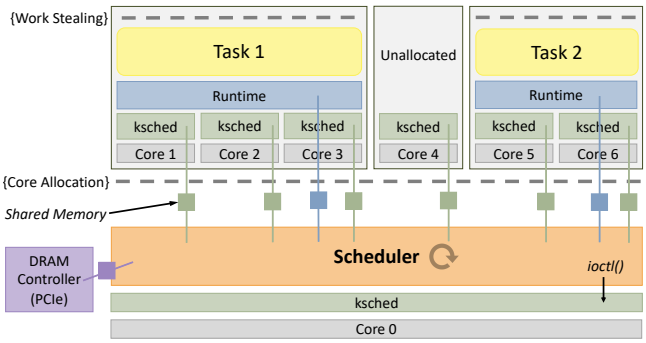


Figure 2: Caladan's system architecture. Caladan relies on a scheduler core to gather control signals from shared memory regions (provided by KSCHED, runtimes, and the DRAM controller). It uses these control signals to adjust core allocations via KSCHED.

the compute capacity lost to interference. Although this cannot fully protect service times, it can prevent queueing delays.

Finally, Caladan introduces a Linux Kernel module called KSCHED. KSCHED performs scheduling functions across many cores at once in a matter of microseconds, even in the presence of interference. KSCHED achieves these goals with three main techniques: (1) it runs on all cores managed by Caladan and shifts scheduling work away from the scheduler core to cores running tasks; (2) it leverages hardware support for multicast interprocessor interrupts (IPIs) to amortize the cost of initiating operations on many cores simultaneously; and (3) it provides a fully asynchronous scheduler interface so that the scheduler can initiate operations on remote cores and perform other work while waiting for them to complete.

5 Design

5.1 Overview

Figure 2 presents the key components of Caladan and the shared memory regions between them. Caladan shares some architectural and implementation building blocks with Shenango [47]: each application is linked with a runtime system, and a dedicated scheduler core (run with root privileges) busy polls shared memory regions to gather control signals and make core allocations. Both systems are designed to interoperate in a normal Linux environment, potentially managing a subset of available cores.

Despite these commonalities, Caladan adopts a radically different approach to scheduling and relies on different scheduling mechanisms. Shenango uses queueing delay as its only control signal to manage changes in load; Caladan uses multiple control signals to manage several types of interference as well as changes in load. Moreover, Shenango's scheduler core combines network processing with CPU scheduling; Caladan's scheduler core is only responsible for CPU scheduling, eliminating packet processing bottlenecks (§6). Finally, Shenango relies on standard Linux system calls to allocate cores, limiting its scalability; Caladan uses KSCHED to more efficiently perform its scheduling functions, including preempting tasks, assigning cores to tasks, detecting when tasks have yielded voluntarily, and reading performance counters from remote cores.

Caladan's runtimes share many properties with those of Shenango. Applications managed by Caladan run inside normal Linux processes, which we refer to as *tasks*. Within each task, the runtime provides "green" threads (light-weight, user-level threads) and kernel-bypass I/O (networking and storage). Runtimes use work stealing to balance load across the cores that are allocated to them—a best practice for minimizing tail latency [51]—and yield cores when they run out of work to steal. Handling threading and I/O in userspace makes managing interference easier in two ways. First, by performing all processing inside the task that needs it, we can better manage the resource contention it generates. By contrast, the Linux Kernel handles I/O on behalf of its tasks, making it difficult to attribute resource usage or interference to a specific task. Second, we can easily instrument the runtime system to export the right per-task control signals (discussed further in §5.2).

Provisioning cores: Users provision each task with a discrete number of *guaranteed cores* (zero or more) that are always available when needed. They can also allocate tasks additional *burstable cores* beyond the number guaranteed, allowing them to make use of any idle capacity. Additionally, each task is designated as LC or BE. BE tasks operate at a lower priority: they are only allocated burstable cores when LC tasks do not need them, they are always provisioned zero guaranteed cores, and they are throttled as needed to manage interference.

In some configurations, it may not be possible to manage interference without harming the performance of LC tasks. To prevent these cases, we recommend a configuration that leaves a small number of cores that are not guaranteed to any task, providing enough slack to manage interference. Caladan can also detect when provisioning constraints prevent it from mitigating interference. As a last resort, this information could be reported back to the cluster scheduler so that it could migrate tasks to other machines. A rich body of prior work has explored adding similar types of interference coordination, as well as identifying complementary workloads, at the cluster scheduler layer [12, 15, 16, 41, 69, 71].

5.2 The Caladan Scheduler

Figure 3 shows the scheduler's key components, the control signals they each use, and their interactions. Separate controller modules detect memory bandwidth and hyperthreading interference, each placing constraints on how cores can be allocated and revoking cores as necessary. The memory bandwidth controller restricts how many cores can be assigned to a task, while the hyperthread controller bans cores within sibling pairs. A top-level core allocator incorporates these restrictions and decides when to grant additional cores to tasks. It tries to minimize queueing delay (to manage changes in load and any unmitigated interference), allocating cores to tasks in a way that respects constraints from the controllers and each

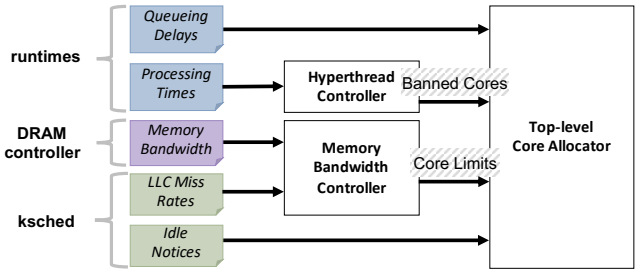


Figure 3: The flow of information through Caladan's scheduler. Control signals flow from runtimes, the DRAM controller, and KSCHED to the controllers and top-level allocator. The hyperthread and memory bandwidth controllers impose constraints on which and how many cores the top-level allocator may grant.

task's resource configuration (the number of guaranteed cores, BE vs. LC, etc.). The controllers and core allocator run once every 10 μs on the scheduler's dedicated core. Because the scheduler can reallocate cores so quickly, it is possible to allocate fractional cores to tasks on average over time (e.g., when less than a full core is needed to accommodate load). This is made efficient through KSCHED's scheduling optimizations (§5.3).

The scheduler gathers control signals from three sources. First, runtimes provide information about request processing times and about queueing delays. Second, the DRAM controller provides information about global memory bandwidth usage. Third, KSCHED provides information about per-core LLC miss rates and notifies the scheduler core when a task has yielded voluntarily. We now discuss each component in more detail. We present each algorithm as synchronous code for clarity, but to handle many tasks concurrently without delaying the scheduler, all code is asynchronous in practice. Each algorithm relies on one tunable parameter; these are described in more detail in Appendix A.

5.2.1 The Top-level Core Allocator

The goal of the top-level core allocator is to grant more cores to tasks that are experiencing queueing delays, whether these delays are due to lingering interference (as shown in the rightmost column of Table 2) or due to changes in load. Algorithm 1 shows its basic operation. The core allocator periodically checks the queueing delay of each task, and, when permitted by the memory bandwidth controller, tries to add cores to the tasks that have delays above a configurable per-task threshold (`THRESH_QD`). Queueing can occur in each runtime core's green thread runqueue, network ingress queue, storage completion queue, and timer heap. Each queued element contains a timestamp of its arrival time, and all queues are placed in shared memory. `QueueingDelay()` computes the delay for each core by summing the delays experienced by the oldest element in each of its queues. It then reports the maximum delay observed across the task's cores.

When a task's delay exceeds its `THRESH_QD`, the allocator

```
1  while True:
2      for each task T:
3          if QueueingDelay(T) < THRESH_QD[T]:
4              continue;
5          if T is limited by BW controller:
6              continue;
7          // try to allocate a core
8          for each core C:
9              if C is banned by HT controller:
10                 continue;
11             if task_on_core[C] has priority over T:
12                 continue;
13             score[C] = CalculateScore(C, T);
14         find core C with highest score;
15         allocate C to T (if found);
16     sleep(10 µs);
```

Algorithm 1: The top-level core allocator.

loops over all cores, checking which cores are allowed by the hyperthread controller and checking which tasks are running on each core. An idle core can be allocated to any task, but a busy core can only be preempted if the core provisioning configuration allows it. For example, if an LC task is only using guaranteed cores, it cannot be preempted by another task. Moreover, a BE task can never preempt an LC task.

Finally, `CalculateScore()` assigns a score to each core, and the core allocator picks the allowed core with the highest score (if one is found). Our scoring function is based on three factors (in order of priority). First, we prefer sibling pairs that are both idle because they have no hyperthreading interference. Second, we prefer hyperthread pairings between *different* tasks because hyperthreading is most efficient when tasks have different performance bottlenecks [31, 45]. Finally, we optimize for temporal locality: Caladan keeps track of the time each task last used each core, and gives the most recent timestamp the highest score. Timestamps are shared between hyperthread siblings, reflecting their shared cache resources.

The core allocator also receives notifications from KSCHED whenever a runtime yields a core voluntarily (not shown in Algorithm 1). When this happens, it updates the `task_on_core` array and immediately tries to grant the core to another task, reducing the cycles the core spends idling.

5.2.2 The Memory Bandwidth Controller

Algorithm 2 shows our memory bandwidth controller. Our aim is to use the majority of available memory bandwidth while avoiding saturation. The memory bandwidth controller periodically polls the DRAM controller's global memory bandwidth usage counter, calculating the access rate since the last polling interval, and triggers when it crosses a saturation threshold (`THRESH_BW`). It then attributes memory bandwidth usage to a specific task by relying on KSCHED to efficiently sample LLC misses from the performance monitoring unit (PMU) [25] of each scheduled core. We found that LLC misses are a good indicator of overall memory bandwidth

```
1  while True:
2      if GlobalMemBandwidth() < THRESH_BW:
3          increment the core limit on the most limited task;
4          sleep(10 µs);
5          continue;
6      for each core C:
7          start[C] = ReadLLCMisses(C);
8      sleep(10 µs);
9      for each core C:
10         end = ReadLLCMisses(C);
11         misses[task_on_core[C]] += end - start[C];
12     find task T that is BE and has the highest misses;
13     decrement task T's core limit and revoke a core;
```

Algorithm 2: The memory bandwidth controller.

```
1  while True:
2      for each core C:
3          T = task_on_core[C]
4          if T is LC and now - GetRequestStartTime(C) ≥
5            THRESH_HT[T]:
6              ban sibling of C;
7          else:
8              unban sibling of C;
9      sleep(10 µs);
```

Algorithm 3: The hyperthread controller.

usage, with the exception that they exclude non-temporal memory accesses, which don't allocate lines in the cache. Fortunately, these are rarely used, but we recommend they be counted in future CPUs. Waiting 10 μs between samples is enough to accurately estimate LLC misses. The bandwidth controller revokes one core from the worst offending task every time it runs until memory bandwidth is no longer saturated. When one task is throttled, another task (that consumes less memory bandwidth) can still use the throttled core.

While Algorithm 2 summarizes this controller's basic behavior, we had to take extra steps to improve its accuracy. First, because `ReadLLCMisses()` initiates PMU counter reads with IPIs (see §5.3), there can be timing skew. Therefore, KSCHED includes the local timestamp counter (TSC), which is stable across cores, when it stores PMU results. This allows us to calculate an LLC miss rate instead of a raw miss count. Second, we discard samples from tasks that have yielded or have been preempted during the measurement interval.

5.2.3 The Hyperthread Controller

Caladan's hyperthread controller detects hyperthread interference and then bans use of the sibling hyperthread until the current request completes (Algorithm 3). Runtimes place timestamps in shared memory to indicate when each hyperthread begins handling a green thread. The hyperthread controller then uses `GetRequestStartTime()` to retrieve these timestamps and check if the current thread has been running for more than a per-task processing time threshold (`THRESH_HT`).

When the threshold has been exceeded, the controller bans use of the sibling hyperthread via KSCHED. The sibling's runtime receives a request from KSCHED to preempt the core and places the current green thread back into its runqueue. The top-level core allocator can detect this as an increase in queueing delay and add back a different (not banned) core. Then KSCHED places the sibling in the shallow `C1` idle state using the `mwait` instruction; `mwait` parks the local hyperthread and reallocates shared physical core resources to the sibling, increasing its performance.

Caladan's hyperthread controller benefits from global knowledge. First, it will only ban a sibling that is handling an LC task if that LC task can be allocated another core, to avoid degrading throughput under high load. Second, if there are not enough cores available, it will prioritize speeding up the green threads that have spent the most time processing a request, keeping tail latency as low as available compute capacity permits. The hyperthread controller can also unban cores, respecting the same priority, when the top-level core allocator needs to allocate a guaranteed core, but none are available due to bans.

Caladan's approach to managing hyperthread interference was inspired by Elfen Scheduling [70]. Our policy for identifying interference is similar to Elfen's refresh budget policy, and both use `mwait` to idle hyperthreads. However, Caladan's approach differs in two key ways. First, Elfen relies on trusted BE tasks to measure interference and yield voluntarily, while Caladan's scheduler makes and enforces these decisions, leveraging the benefits of global knowledge. Second, Elfen can only support pinning one LC task and one BE task to each hyperthread pair. Instead, we allow any pairing (even self pairings) and can handle interference between LC tasks. This enables significantly higher throughput because all logical cores are available for use by any task (§7.3).

5.2.4 An Example: Reacting to Garbage Collection

As an example, we explain how Caladan's scheduler responds when a GC cycle begins, causing memory bandwidth interference for an LC task (the workload depicted in Figure 1). As soon as global memory bandwidth usage exceeds `THRESH_BW`, the memory bandwidth controller will revoke cores from the GC task, revoking one core every 10 μs until total memory bandwidth usage falls below `THRESH_BW` (Algorithm 2). In the meantime, the LC task may suffer from interference, increasing its queueing delay. This will cause the top-level core allocator to grant it additional cores, beginning with any idle cores, but preempting additional cores from the GC task if necessary. It will add one core every 10 μs until the LC task's queueing delay falls below its `THRESH_QD` again (Algorithm 1). Once the GC interference has been successfully mitigated, the LC task will yield the extra cores.

5.3 KSCHED: Fast and Scalable Scheduling

KSCHED's goal is to efficiently expose control over CPU scheduling to the userspace scheduler core. A scheduler core

that relies on the current Linux Kernel system call interface is subject to its limitations; KSCHED must overcome these. First, Linux system calls, like `sched_set_affinity()`, perform computationally expensive work (e.g., locking runqueues) on the core that calls them. Second, Linux system calls block and reschedule while waiting for their operation to complete, preventing the scheduler core from performing other work. Third, Linux system calls can only perform one operation at a time, squandering any opportunity to amortize costs across multiple operations and cores. Finally, cores may only directly read their own performance counters and Linux provides no efficient mechanism to query those on other cores.

KSCHED adopts a radically different approach from Linux's existing mechanisms, supporting direct communication between the scheduler core and kernel code running on other cores via per-core, shared-memory regions. The scheduler core writes commands into these regions and then uses an `ioctl()` to *kick* the remote cores by sending them IPIs. KSCHED then executes the commands (in kernelspace on the remote cores) and writes back results.

KSCHED supports three commands: waking tasks (potentially preempting the current task), idling cores, and reading performance counters. Before preempting a task or idling a core, KSCHED delivers a signal to the runtime to give it a few microseconds to yield cleanly, saving the current green thread's register state and placing it back in the runqueue. Then, to wake a new task on a core, KSCHED locks the task's affinity so that Linux cannot migrate it to another core and calls into the Linux scheduler. To idle a core instead, KSCHED calls `mwait`. Finally, KSCHED can sample any performance counter on any core, and includes the TSC in the response.

When the scheduler kicks a core, the IPI handler immediately processes any pending commands. Commands can also be processed without IPIs by cores that are idle through efficient polling. To achieve this, KSCHED bypasses the standard Linux idle handler, setting a flag that notifies the scheduler core that the current task has yielded voluntarily. KSCHED then checks for new commands; if none are available, it runs the `monitor` instruction, telling the core to watch the cache line containing the shared region. Finally, it parks the core with the `mwait` instruction, placing it in the shallow `C1` idle state. `mwait` monitors cache coherence messages and immediately resumes execution when the shared region is written to by the scheduler core.

One of the most expensive operations that both Linux and KSCHED must perform is sending IPIs. When there are multiple operations, KSCHED leverages the multicast capability of the interrupt controller to send multiple IPIs at once, significantly amortizing costs. To facilitate this, the scheduler core writes all pending operations to shared memory and then passes a list of cores to kick to an `ioctl()` that initiates IPIs. In addition, all of KSCHED's commands are issued asynchronously, so that the scheduler core can perform other work while waiting for them to complete. Finally, KSCHED performs expensive operations such as sending signals and affinitizing tasks to cores on the targeted cores rather than on the scheduler core. In combination, these three properties allow KSCHED to perform scheduling operations with low overhead, enabling Caladan to support high rates of core reallocation and performance counter sampling even with many concurrent tasks (§7.3).

6 Implementation

Caladan is derived from the open-source release of Shenango [61], but we implemented a completely new scheduler and the KSCHED kernel module, which are 3,524 LOC and 533 LOC, respectively. Shenango was a good starting point for our system because of its feature-rich runtime with support for green threads and TCP/IP networking. Moreover, Shenango's runtime is already designed to handle signals to cleanly preempt cores [47].

We modified Shenango's runtime in two important ways. First, Shenango relies on its scheduler core to forward packets in software to the appropriate runtime over shared memory queues. Instead, we linked the `libibverbs` library directly into each runtime, providing fast, kernel-bypass access to networking. This implementation strategy allowed us to completely eliminate the packet forwarding bottlenecks imposed by Shenango and also reduced our scheduler core's exposure to interference, by reducing its memory and computational footprint. Our scheduler core measures packet queueing delay by mapping the NIC's RX descriptor queues (for each task) over shared memory and accessing the packet arrival timestamps encoded in the descriptors by the NIC. Second, we augmented the runtime with support for NVMe storage using Intel's SPDK library to bypass the kernel. These changes required us to add 2,943 new LOC to the runtime, primarily to add integration with `libibverbs` and SPDK.

To support idling in KSCHED, each per-core shared memory region uses a single cache line (64 bytes) because `mwait` can only monitor regions of this size. We packed these cache lines into a contiguous array so that our scheduler core could take advantage of hardware prefetching to speed up polling. KSCHED allows the scheduler core to control which idle state `mwait` enters, but we have not yet explored power management. We also modified the Linux Kernel source to accelerate multicast IPIs; although the Linux Kernel provides an API called `smp_call_function_many()` that supports this feature, it imposes additional software overhead, especially under heavy memory bandwidth interference.

7 Evaluation

We evaluate Caladan by answering the following questions:

1. How does Caladan compare to previous systems (§7.1)?
2. Can Caladan colocate different tasks while maintaining low tail latency and high CPU utilization (§7.2)?

3. How do the individual components of Caladan's design enable it to perform well (§7.3)?

Experimental setup: We evaluate our system on a server with two 12 physical core (24 hyperthread) Xeon Broadwell CPUs and 64 GB of RAM running Ubuntu 18.04 with kernel 5.2.0 (modified to speed up multicast IPIs). We do not consider NUMA, and direct all interrupts, memory allocations, and threads to the first socket. The server is equipped with a 40 Gb/s ConnectX-5 Mellanox NIC and a 280 GB Intel Optane NVMe device capable of performing random reads at 550,000 IOPS. To generate load, we use a set of quad-core machines with 10 Gb/s ConnectX-3 Mellanox NICs connected to our server via a Mellanox SX1024 non-blocking switch. We tune the machines for low latency in accordance with recommended practices, disabling TurboBoost, CPU idle states, CPU frequency scaling, and transparent hugepages [37]. We also disable Meltdown [2] and MDS [24] mitigations, since these vulnerabilities have been fixed by Intel in recent CPUs. When evaluating Linux's performance, we run BE tasks with low-priority using `SCHED_IDLE` and use kernel version 5.4.0 to take advantage of recent improvements to `SCHED_IDLE`. We use *loadgen*, an open-loop load generator, to generate requests with Poisson arrivals over TCP connections [61]. Unless stated otherwise, we configure all Caladan experiments with 22 guaranteed cores for LC tasks, leaving one physical core for the scheduler.

Evaluated applications: We evaluate three LC tasks. First, *memcached* (v1.5.6) is a popular in-memory, key-value store that has been extensively studied [43]. We generate a mix of reads and writes based on Facebook's `USR` request distribution [6] (service times of about 1 μs). Second, *silo* is a state-of-the-art, in-memory, research database [64]. We feed it the TPC-C request pattern, which has high service time variability (20 μs median; 280 μs 99.9%-ile) [63]. Silo is only a library, so we integrated it with a server that can handle RPCs, performing one transaction per request. Finally, we built a new NVMe block storage server inspired by ReFlex [34], that we call *storage*. We added compression (using Snappy [1]) and encryption (using AES-NI [46]) to study the hyperthreading effects of RPC frameworks that rely on vector processing. We preload the SSD with XML-formatted data from Wikipedia [39], and issue requests for blocks of varying lengths (99% 4KB, 1% 44KB) to evaluate service time variability (35 μs and 250 μs for each respective size).

For BE tasks, we use workloads from the PARSEC benchmark suite [9]. In particular, we evaluate *x264*, an H.264/AVC video encoder, *swaptions*, a portfolio pricing tool, and *streamcluster*, an online clustering algorithm. We modified swaptions to use the Boehm garbage collector to allocate its memory objects [10], allowing us to study the interference caused by garbage collection; we call this version *swaptions-GC*. All three workloads exhibit phased behavior, changing their resource usage over regular intervals (some have much larger variance than others). Finally, we evaluate a synthetic antagonist that continuously reads and writes arrays of memory in two configurations: *stream-L2* displaces the L2 cache, while *stream-DRAM* displaces the LLC and consumes all available memory bandwidth.

All applications run in our modified Shenango runtime, which supports standard abstractions such as TCP sockets and the pthread interface (via a shim layer), making it relatively straightforward to port and develop applications (§8).

Parameter tuning: Caladan has three parameters that are user-tunable and can make tradeoffs between latency and CPU efficiency. Appendix A explains how to tune these parameters and shows how sensitive Caladan's performance is to particular choices of these parameters. In our evaluation, we tuned all three for low latency. First, we set `THRESH_QD` (the queueing delay threshold) to 10 μs for all tasks. Second, we set `THRESH_BW` (the memory bandwidth threshold) to 25 GB/s. Finally, we set a `THRESH_HT` (the processing time threshold) for each LC task (not supported for BE tasks). We set it to 25 μs for silo, 40 μs for storage, and infinite for memcached.

Comparison with Parties: Parties [12] is the most relevant prior work for mitigating interference. It builds upon Heracles [38] by adding support for multiple LC tasks. Ideally, we would compare directly to Parties, but its source code is not publicly available, and we were unable to obtain it from the Parties authors. Instead, we reimplemented Parties in accordance with the details described in its paper.

By implementing Parties ourselves, we were able to use the same runtime system for both Caladan and Parties, so they could benefit equally from kernel-bypass I/O, allowing us to evaluate only differences in scheduling policy. We did not implement some components in Parties that were not relevant to our experiments. Specifically, our workloads do not contend over disk, network, or memory capacity. Managing these resources is important but unrelated to our focus on CPU interference. Moreover, we did not include the CPU frequency scaling controller, as reducing energy consumption is outside the scope of our work. We did implement all of Parties' key mechanisms, including core allocation, CAT, and an external measurement client that samples tail latencies over 500 ms periods. We also invested considerable effort in tuning Parties' latency thresholds to yield the best possible performance.

Normally, Parties leaves hyperthreads disabled because it is unable to manage this form of interference, reducing its CPU throughput. Instead, we enabled hyperthreads with a policy that prefers self pairings. For specifically memcached—the workload we evaluated—this forms a complementary pairing that has minimal effect on latency, but allowed us to conduct a direct comparison with the same number of cores. The addition of kernel-bypass networking and hyperthread pairing enable our version of Parties to significantly outperform the reported performance of the original, so we refer to it as *Parties**.

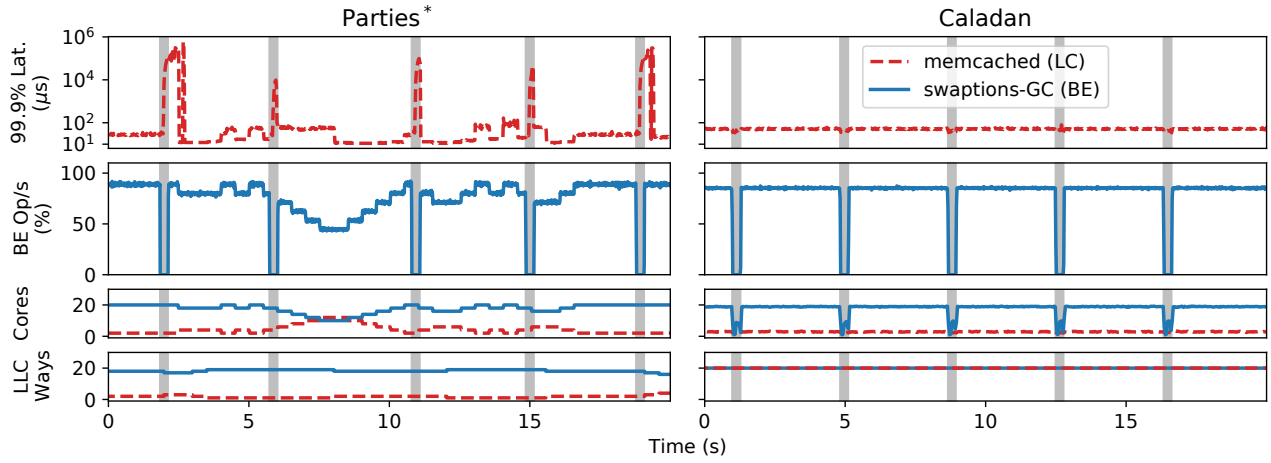


Figure 5: Timeseries of memcached colocated with a garbage-collected BE task for Parties* (left) and Caladan (right). Gray bars indicate GC cycles in the BE task. The Parties* resource controller algorithm is unable to provide performance isolation and high CPU utilization when tasks have dynamic resource demands, while Caladan maintains both. Top graphs have log-scaled y-axes.

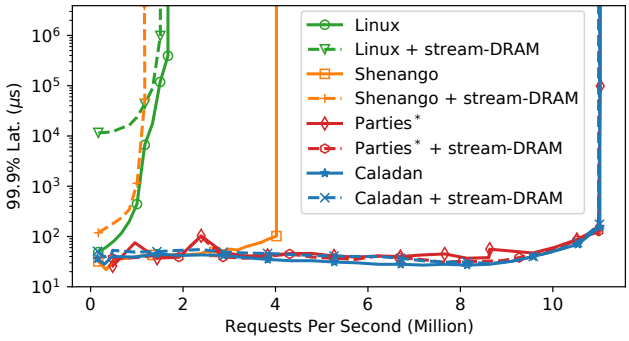


Figure 4: Constant memory bandwidth interference degrades memcached performance for Linux and Shenango, but Caladan and Parties* can mitigate it. Note the log-scaled y-axis.

7.1 Comparison to Other Systems

Constant interference: To demonstrate the necessity of managing interference, we first compare Parties* and Caladan to systems that do not explicitly manage interference. We evaluate a relatively less challenging scenario, where an LC task (memcached) is colocated with stream-DRAM, a BE task that generates constant memory bandwidth interference.

Figure 4 illustrates that, as expected, both Linux and Shenango suffer significant increases in tail latency in the presence of colocation, reaching tail latencies up to 235× and 6× higher than without interference, respectively. Shenango's throughput also decreases by 75% in the presence of interference, because its scheduler core becomes overloaded with packet processing, due to higher cache miss rates and memory access latency caused by stream-DRAM. In contrast, Caladan and Parties* are both able to maintain similar tail latency with and without interference, because they manage it explicitly. Both also achieve much higher throughput than Linux and Shenango because runtime cores send and receive packets directly using our runtime's kernel-bypass network stack (§6), preventing the Linux network stack or the scheduler core from becoming a bottleneck. While adapting Shenango to use our runtime's kernel-bypass network stack would eliminate this throughput bottleneck, it would not improve the tail latency of LC tasks suffering from interference.

Phased interference: We now focus on interference caused by phased behavior, a more difficult and realistic case that Caladan is designed to solve. We revisit the garbage collection experiment from Section §2, colocating an instance of memcached with swaptions-GC. We issue 800,000 requests per second to memcached for a period of 120 seconds and measure its tail latency over 20 ms windows. We show the first 20 seconds of the experiment in Figure 5, which we found to be representative of the behavior during the entire experiment.

Caladan throttles the BE task's cores as soon as each GC cycle starts, preventing latency spikes, and it gives back cores to the BE task as soon as the GC cycle ends, maintaining high BE throughput. Parties* attempts to find an allocation of cores and cache ways that minimizes latency and maximizes resources for the BE task, but it is unable to converge when resource demands are shifting at timescales much smaller than its 500 ms adjustment interval. Often Parties* grants additional cores in response to GC cycles, but these adjustments happen too slowly to prevent latency spikes. As a result, Parties* experiences 99.9% latency that is 11,000× higher than Caladan during GC cycles. In addition, Parties* also harms BE throughput, achieving an average of 5% less than Caladan because it punishes swaptions-GC by too much and for too long. These results show that faster reaction times are essential when handling tasks with phased behaviors.

7.2 Diverse Colocations

Two tasks: To understand if Caladan can maintain its benefits in diverse situations, we evaluate 15 colocations between pairs of LC and BE tasks with different resource usages, service

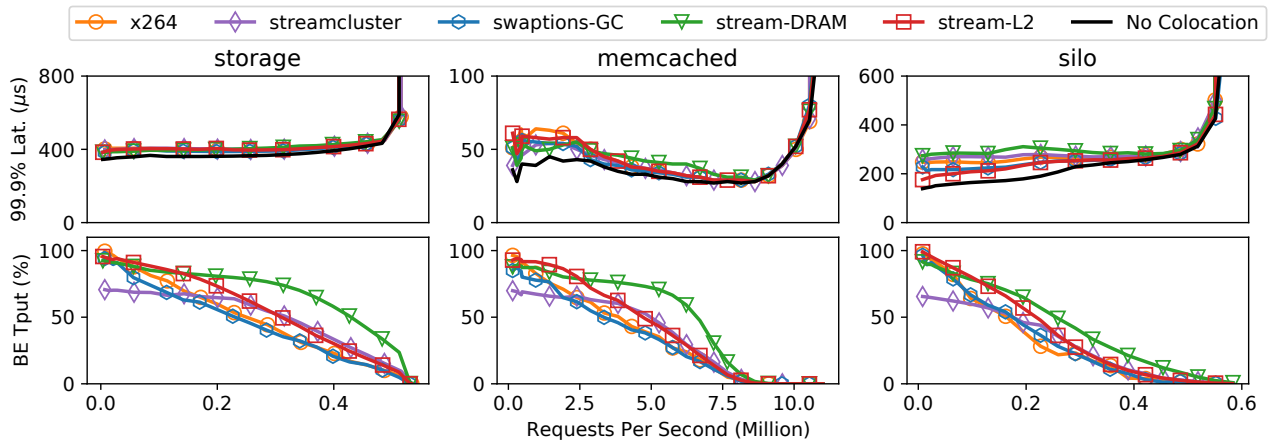


Figure 6: Caladan can colocate many combinations of LC and BE applications with only modest latency penalties for LC tasks (top), while maintaining excellent throughput for BE tasks (bottom).

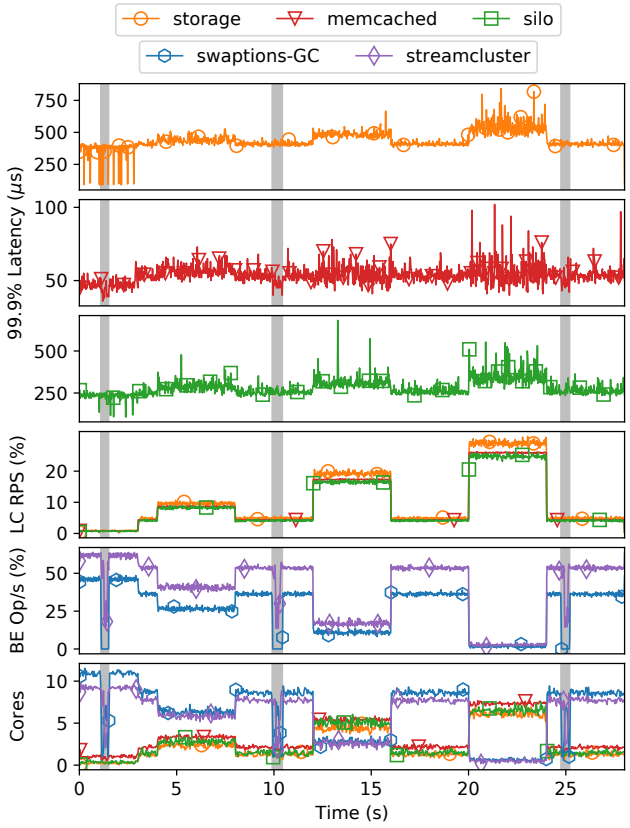


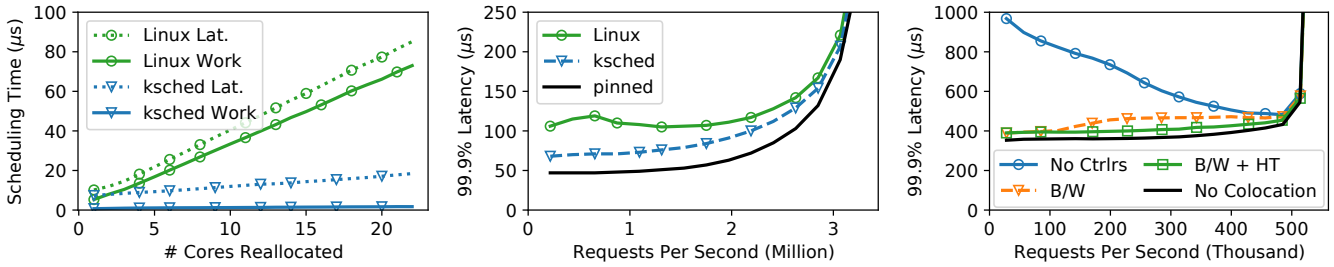
Figure 7: Caladan can colocate multiple LC and BE apps while providing performance isolation and high utilization. Gray bars indicate GC cycles in swaptions-GC.

time distributions, and throughputs. 9 out of 15 pairings include BE tasks with phased behaviors. We consider the impact on each LC task's tail latency and the amount of throughput the BE task can achieve by using burstable cores.

In this experiment (Figure 6), each data point represents a different fixed average load offered to an LC task (columns), while it is paired with a BE task (colors/linetypes). Caladan is highly effective at mitigating interference: storage and memcached achieve nearly the same tail latency as they do without colocation. Silo experiences a small increase in tail latency at low load because it is sensitive to LLC interference, leading to service time but not queueing delay increases. At higher load, silo generates self interference, so it experiences similar tail latency with and without colocation. Overall, Caladan can easily maintain microsecond-level tail latency under challenging colocation conditions.

At the same time, Caladan yields excellent BE task throughput. The exact BE throughput depends on the degree of resource contention with the LC task. For example, x264, swaptions-GC, and stream-L2 use less memory bandwidth (on average), so they can linearly trade CPU time with the LC task. Streamcluster and stream-DRAM both consume a larger amount of memory bandwidth, so they are throttled by our memory bandwidth controller. However, they also pair well with LC tasks as siblings (especially memcached) because they use different physical core resources. At higher LC load, these BE tasks are given fewer cores so they use less memory bandwidth and are then throttled less. Overall, BE throughput depends on the specific interactions between the BE and LC tasks, and varies with LC load. To the best of our knowledge, Caladan is the first system to achieve both microsecond-level LC tail latency and high BE throughput under such a broad range of conditions.

Many tasks: To demonstrate Caladan's ability to manage many tasks simultaneously, we colocate all 3 of the LC tasks along with swaptions-GC and streamcluster (each LC task is configured with 6 guaranteed cores). Figure 7 shows a 30-second trace from this experiment, during which the load of each of the LC apps changes multiple times (4th graph) and swaptions-GC performs garbage collection three times (gray bars). When load or interference changes, Caladan converges nearly instantly. When GC is not running, the combination of streamcluster and swaptions-GC does not saturate memory bandwidth. However, when GC begins, both tasks together saturate DRAM bandwidth and are throttled by Caladan. Cal-



(a) Scheduling Scalability **(b)** Aggregated Latency of 11 LC Apps **(c)** Contribution of Sub-controllers

Figure 8: (a) KSCHED dramatically increases scheduling scalability over Linux. (b) Aggregated latency for 11 LC apps scheduled using KSCHED vs. Linux mechanisms. (c) The impact of each Caladan sub-controller on tail latency for storage paired with stream-DRAM.

adan's fast reactions (up to 230,000 core reallocations per second) enable all three LC tasks to maintain low tail latency (top three graphs) throughout constantly shifting load and interference.

7.3 Microbenchmarks

KSCHED: To evaluate the benefits of KSCHED's faster scheduling operations, we run a simple microbenchmark where we continuously rotate tasks to different cores. To measure scalability, we migrate different numbers of tasks together in groups. We run the benchmark both with KSCHED and with a variant that uses standard Linux system calls such as `sched_setaffinity()`, `tgkill()`, and `eventfd()`.

Figure 8a shows both the scheduling work (time spent by the scheduler core) and the scheduling latency (time until the migration completes) per migration. Both metrics benefit tremendously from KSCHED's multicast IPIs, allowing it to amortize the cost of multiple simultaneous migrations. By contrast, Linux's system call interface suffers from overhead and because it cannot support batching; operations must be serialized, increasing scheduling work by 43× and scheduling latency by 5× when moving 22 tasks. In addition, KSCHED maintains low scheduling work even with many tasks by offloading expensive operations such as sending signals to remote cores.

We demonstrate the value of these improvements in an experiment with 11 synthetic LC tasks and 2 synthetic non-interfering BE tasks. The LC tasks have 5 μs average service times that are exponentially distributed and each is configured with 2 guaranteed cores. We compare against an earlier version of Caladan that employed the Linux scheduling mechanisms evaluated above. In Figure 8b, we show that Caladan is able to maintain much lower tail latency for the LC tasks (close to that of running with cores pinned). In this experiment, Caladan performs up to 560,000 core reallocations per second at its peak (at a load of 0.65 million RPS), while the version using Linux mechanisms bottlenecks at around 285,000 allocations per second. KSCHED provides similar benefits for sampling performance counters (not shown).

Controllers: We found that both the memory bandwidth and hyperthread controllers were necessary in order to ensure

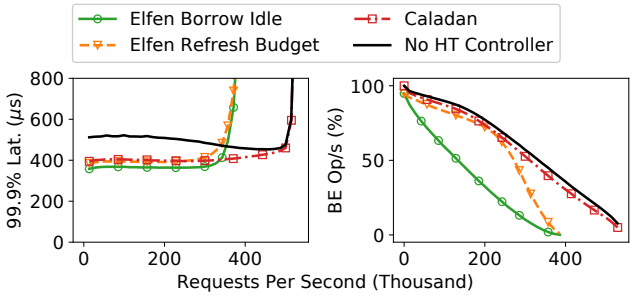


Figure 9: Caladan enables higher LC throughput than Elfen by allowing arbitrary tasks to co-run on a physical core (including the same LC task).

isolation across a variety of tasks and loads. To provide one concrete example, Figure 8c evaluates the contribution of each controller module to the storage LC task when colocated with the stream-DRAM BE task. At very low loads, the bandwidth controller is sufficient to provide low tail latency. This is because as Caladan revokes cores from the BE task, it leaves the hyperthread pair cores of the LC task idle, rendering the hyperthread controller unnecessary. However, at higher LC loads, both controllers are necessary in order for the storage task to achieve nearly the same tail latency as it would have without colocation.

Next we focus on the hyperthread controller and evaluate the benefits of allowing any two tasks to co-run on a physical core (e.g., two LC tasks or even two hyperthreads in the same task). Figure 9 compares Caladan to two modified versions of Caladan that implement Elfen's [70] scheduling policies, when colocating storage and stream-L2. Elfen's borrow idle policy disallows co-running, only allowing the BE to run on a physical core when it is not being used by the LC; this yields low tail latency for the LC task but also low BE throughput. Elfen's refresh budget policy, which Caladan generalizes (§5.2.3), yields higher BE throughput at the cost of a slight increase in tail latency, demonstrating the benefits of using both hyperthreads simultaneously. Caladan achieves 37% more LC throughput than Elfen by enabling the LC task to co-run with itself. Similarly, at low LC loads, Caladan is able to achieve 5% higher BE throughput than Elfen since BE tasks can use both hyperthread lanes. Finally, running Caladan with the hyperthread controller disabled yields slightly higher BE

throughput but at a cost of up to 117 μs higher tail latency, highlighting the need to explicitly manage hyperthread interference to achieve both high throughput and low tail latency.

8 Discussion

Compatibility: Caladan requires applications to use its runtime system because it depends on it to export control signals to the scheduler, and to rapidly map threads and packet processing work across a frequently changing set of available cores. Our runtime is not fully Linux compatible, but it provides a realistic, concurrent programming model (inherited from Shenango) that includes threads, mutexes, condition variables, and synchronous I/O [47]. Caladan also includes a partial compatibility layer for system libraries (e.g., libpthread) that can support PARSEC [9] without modifications, giving us some confidence our design is flexible enough to support unmodified Linux applications in the future. Applications that do not use our runtime can coexist on the same machine, but they must run on cores that are not managed by Caladan, and they cannot be throttled if they cause interference.

The more fundamental requirement for Caladan is the need for LC tasks to expose their internal concurrency to the runtime (e.g., by spawning green threads), potentially requiring changes to existing code. If there is insufficient concurrency, a task will be unable to benefit from additional cores, hindering Caladan's ability to manage shifts in load or interference. In general, we recommend that tasks expose concurrency by spawning either a thread per connection or a thread per request. For example, normally memcached multiplexes multiple TCP connections per thread, but we modified it to instead spawn a separate thread to handle each TCP connection.

On the other hand, Caladan can support BE tasks that do not expose their internal concurrency, as it can still throttle them if they cause too much interference. For example, if a BE task is single-threaded (i.e., has no concurrency), and it consumes too much memory bandwidth, Caladan will oscillate between giving it one and zero cores, effectively time multiplexing its memory bandwidth usage. However, BE tasks can optionally achieve higher performance by exposing their internal concurrency: load will be more evenly balanced and they will be able to take advantage of burstable cores.

Limitations: Our current implementation of Caladan has two limitations. First, it is unable to manage interference across NUMA nodes. NUMA introduces additional shared resources that are vulnerable to interference, including an inter-socket interconnect and separate memory controllers per node. Fortunately, high-precision performance counters are available for these resources, and we plan to explore NUMA-aware interference mitigation strategies in the future, such as revoking cores or migrating tasks between nodes. Second, our scheduling policies do not minimize the threat of transient execution attacks across hyperthread siblings [3, 11, 65]. Ideally, only mutually-trusting tasks should be allowed to run on sibling cores. At the time of writing, a similar capability is under development for the Linux Kernel [13].

Future work: One promising opportunity for future work is to incorporate hardware partitioning back into Caladan's design. For example, if a BE task uses high memory bandwidth and lacks temporal locality, many of the cache lines it occupies in the LLC will be wasted. Under these conditions, Caladan is still effective at preventing latency increases, but it must allocate extra cores to victim tasks. If future hardware partitioning mechanisms could be designed to accommodate frequently shifting LLC usage—or if static LLC usage could be identified and managed through existing mechanisms—CPU efficiency could be further improved.

9 Related Work

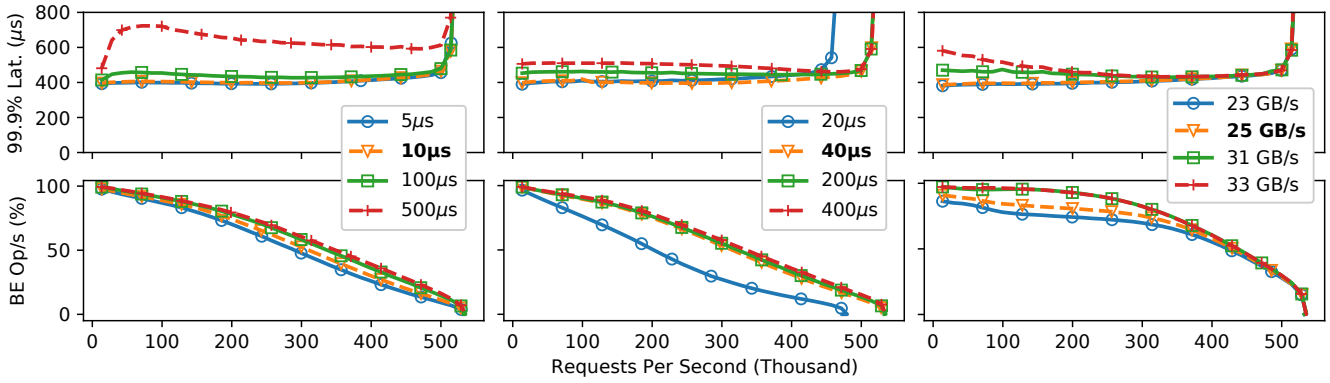
Interference management: Many prior systems manage interference between LC and BE tasks by statically partitioning resources [19, 28, 50, 62]. While this approach can reduce interference, it sacrifices CPU utilization because each task must be provisioned enough resources to accommodate peak load. Heracles [38], Parties [12], and PerfIso [27] instead adjust partitions dynamically. However, unlike Caladan, these systems cannot manage changes in interference while maintaining microsecond latency and high utilization.

Efforts to isolate the network [35, 36, 53] or storage [34] are complementary to Caladan. We do not currently focus on power management [32, 58] or TurboBoost [23], because we optimize for the setting in which all cores are fully utilized, but it should be possible to integrate power management with Caladan to improve its CPU efficiency at lower utilization.

User-level core allocators: To enable low latency in the face of fluctuating load, systems like IX [8], PerfIso [27], Shenango [47], and Arachne [52] introduce user-level core allocators that estimate load and reallocate cores to BE tasks when they are not needed by LC tasks. Similarly, TAS [33] and Snap [42] adjust cores in response to changes in packet processing load. Like these systems, Caladan manages changes in load through core allocations, but it goes a step further by using core allocation to manage interference too.

Scheduling optimizations: Shinjuku [30] proposes fine-grained preemption to reduce tail latency, using Dune [7] to provide fast, direct access to IPIs in userspace. KSCHED includes kernel optimizations that allow for similar performance when sending an IPI to a single core, but it speeds up IPIs over Shinjuku's reported speeds when sending more than one IPI at a time because of its multicast IPI optimization.

Dataplane systems: There has also been significant work on optimizing OS networking for throughput and latency [8, 29, 33, 48, 52, 55]. ZygOS proposes work stealing as a technique to reduce tail latency under variable service times [51]. Arachne [52] and Shenango [47] build a similar latency reduction strategy on top of green threads to improve programmability. Caladan builds upon all of these ideas to eliminate



(a) THRESH_QD **(b)** THRESH_HT **(c)** THRESH_BW

Figure 10: Parameter sensitivity for the storage LC task; the parameters used in our evaluation appear in bold. (a) THRESH_QD allows an operator to achieve better tail latencies at the expense of BE throughput. (b) THRESH_HT reins in the latency of long requests, but setting it too low reduces BE throughput. (c) THRESH_BW is set to avoid exponential increases in memory access latencies.

network processing and queueing bottlenecks, allowing it to manage interference unperturbed by software overheads or load imbalances.

10 Conclusion

This paper presented Caladan, an interference-aware CPU scheduler that significantly improves performance isolation while maintaining high CPU utilization. Caladan's effectiveness comes from its speed: by matching control signals and actions to the same timescale that interference affects performance, Caladan can mitigate interference before it can harm quality of service. Caladan relies on a carefully selected set of control signals to manage multiple forms of interference in a coordinated fashion, and combines a wide range of optimizations to rapidly gather control signals and make core allocations faster. These contributions allow Caladan to deliver microsecond-level tail latency and high CPU utilization while colocating multiple tasks with phased behaviors.

11 Acknowledgments

We thank our shepherd Kathryn S. McKinley, the anonymous reviewers, Frans Kaashoek, Malte Schwarzkopf, Akshay Narayan, and other members of PDOS for their useful feedback. We thank CloudLab [17] and Eitan Zahavi at Mellanox for providing equipment used to test and evaluate Caladan. This work was funded by the DARPA FastNICs program under contract #HR0011-20-C-0089, by a Facebook Research Award, and by a Google Faculty Award.

A Parameter Tuning and Sensitivity

In this Appendix, we describe how to set Caladan's three user-tunable parameters and show how sensitive Caladan's performance is to particular choices of these parameters. To illustrate the behavior of THRESH_QD and THRESH_HT, we colocate the storage workload with stream-L2. For THRESH_BW, we colocate the storage workload with stream-DRAM. In each case, we vary a single parameter, while fixing other parameters to the values used in our evaluation.

THRESH_QD represents the per-task queueing delay limit before the top-level core allocator tries to grant another core. As shown in Figure 10a, an operator can trade some LC tail latency for higher BE throughput using a value of THRESH_QD larger than Caladan's default 10 μs. For example, a THRESH_QD of 100 μs enables 7% more BE throughput at the cost of 54 μs higher LC tail latency for these workloads. We chose to optimize for tail latency, and found that values below 10 μs degraded BE throughput without further improving LC tail latency.

THRESH_HT places a worst-case limit on how long a request can be delayed by a task generating interference on its hyperthread sibling. If it is set too low (i.e., most request processing requires a dedicated physical core), BE throughput will suffer and LC latency will degrade at high load due to insufficient compute capacity. For a skewed service time distribution, like our storage workload, choosing a value above the median is a good heuristic. Figure 10b illustrates that setting THRESH_HT below the median of 35 μs significantly lowers BE throughput, while values that are slightly above the median yield increased BE throughput and good tail latency. For workloads with service times less than 5 μs (e.g., memcached), we recommend setting THRESH_HT to infinite because mwait requires a few microseconds to park a hyperthread.

Finally, THRESH_BW represents the global maximum allowed memory bandwidth usage before Caladan begins to throttle tasks. THRESH_BW should be set once per machine to a bandwidth just low enough to avoid the exponential increase in memory access latency that occurs close to memory bandwidth saturation. We use 25 GB/s for our machine (70–80% of its capacity), which keeps memory latency low for any access pattern. Figure 10c shows this setting trades a small amount of BE throughput in exchange for predictable latency.

B Artifact

Caladan's source code, ported applications, and experiment scripts can be found at https://github.com/shenango/caladan-all.

References

[1] Snappy. https://github.com/google/snappy.

[2] Intel analysis of speculative execution side channels. Technical report, January 2018.

[3] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri. Port contention for fun and profit. In *IEEE S&P*, 2019.

[4] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of response latency on user behavior in web search. In *SIGIR*, 2014.

[5] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *NSDI*, 2018.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[7] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, 2012.

[8] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *TOCS*, 2017.

[9] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[10] H. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *PLDI*, 1991.

[11] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[12] S. Chen, C. Delimitrou, and J. F. Martínez. PARTIES: QoS-aware resource partitioning for multiple interactive services. In *ASPLOS*, 2019.

[13] J. Corbet. Completing and merging core scheduling. https://lwn.net/Articles/820321/, Sept. 2020.

[14] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[15] C. Delimitrou and C. Kozyrakis. QoS-aware scheduling in heterogeneous datacenters with Paragon. *TOCS*, 2013.

[16] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *ASPLOS*, 2014.

[17] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *USENIX ATC*, 2019.

[18] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sánchez. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In *HPCA*, 2018.

[19] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *SIGCOMM*, 2020.

[20] T. Harris, M. Maas, and V. J. Marathe. Callisto: co-scheduling parallel runtime systems. In *EuroSys*, 2014.

[21] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *HPCA*, 2016.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[23] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *HPCA*, 2015.

[24] Intel. Microarchitectural data sampling. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-analysis-microarchitectural-data-sampling.

[25] Intel Corporation. *Intel 64 and IA-32 Architectures Performance Monitoring Events*, December 2017.

[26] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, April 2020.

[27] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. R. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance isolation for commercial latency-sensitive services. In *USENIX ATC*, 2018.

[28] S. A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *SoCC*, 2019.

[29] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *NSDI*, 2014.

[30] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *NSDI*, 2019.

[31] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks. Profiling a warehouse-scale computer. *IEEE Micro*, 2016.

[32] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *MICRO*, 2015.

[33] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP acceleration as an OS service. In *EuroSys*, 2019.

[34] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash $\approx$ local flash. In *ASPLOS*, 2017.

[35] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM*, 2015.

[36] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, et al. PicNIC: predictable virtualized NIC. In *SIGCOMM*. 2019.

[37] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *EuroSys*, 2014.

[38] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *ISCA*, 2015.

[39] M. Mahoney. Large text compression benchmark. http://www.mattmahoney.net/text/text.html, 2011.

[40] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[41] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

[42] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. E. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

[43] Memcached community. memcached – a distributed memory object caching system. https://memcached.org/.

[44] Michael Larabel. Intel hyper threading performance with a Core i7 on Ubuntu 18.04 LTS. https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4, 2018.

[45] J. Nakajima and V. Pallipadi. Enhancements for hyper-threading technology in the operating system: Seeking the optimal scheduling. In *WIESS*, 2002.

[46] OpenSSL. OpenSSL cryptography and SSL/TLS toolkit. https://openssl.org/.

[47] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[48] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *TOCS*, 2015.

[49] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, 2017.

[50] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, 2014.

[51] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *SOSP*, 2017.

[52] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. K. Ousterhout. Arachne: Core-aware thread management. In *OSDI*, 2018.

[53] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: scalable NIC for end-host rate limiting. In *NSDI*, 2014.

[54] S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity in interactive services. In *ICAC*, 2013.

[55] L. Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX ATC*, 2012.

[56] D. Sánchez and C. Kozyrakis. Scalable and efficient fine-grained cache partitioning with Vantage. *IEEE Micro*, 2012.

[57] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[58] E. Sharafzadeh, S. A. S. Kohroudi, E. Asyabi, and M. Sharifi. Yawn: A CPU idle-state governor for data-center applications. In *APSys*, 2019.

[59] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.

[60] The Go Community. The go programming language. https://golang.org.

[61] The Shenango Authors. Shenango's open-source release. https://github.com/shenango/shenango.

[62] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. J. Argyraki, S. Ratnasamy, and S. Shenker. ResQ: Enabling SLOs in network function virtualization. In *NSDI*, 2018.

[63] TPC. TPC-C benchmark. http://www.tpc.org/tpcc/.

[64] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[65] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: rogue in-flight data load. In *IEEE S&P*, 2019.

[66] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[67] R. Wang and L. Chen. Futility scaling: High-associativity cache partitioning. In *MICRO*, 2014.

[68] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[69] H. Yang, A. D. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *ISCA*, 2013.

[70] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX ATC*, 2016.

[71] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.

[72] L. Zhao, R. R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *PACT*, 2007.

[73] H. Zhu and M. Erez. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. In *ASPLOS*, 2016.







Overload Control for µs-Scale RPCs with Breakwater

Inho Cho Ahmed Saeed Joshua Fried Seo Jin Park Mohammad Alizadeh Adam Belay
MIT CSAIL

Abstract

Modern datacenter applications are composed of hundreds of microservices with high degrees of fanout. As a result, they are sensitive to tail latency and require high request throughputs. Maintaining these characteristics under overload is difficult, especially for RPCs with short service times. In this paper, we consider the challenging case of microsecond-scale RPCs, where the cost of communicating information and dropping a request is similar to the cost of processing a request. We present Breakwater, an overload control scheme that can prevent overload in microsecond-scale services through a new, server-driven admission control scheme that issues credits based on server-side queueing delay. Breakwater contributes several techniques to amortize communication costs. It engages in demand speculation, where it assumes clients have unmet demand and issues additional credits when the server is not overloaded. Moreover, it piggybacks client-side demand information in RPC requests and credits in RPC responses. To cope with the occasional bursts in load caused by demand speculation, Breakwater drops requests when overloaded using active queue management. When clients' demand spikes unexpectedly to 1.4× capacity, Breakwater converges to stable performance in less than 20 ms with no congestion collapse while DAGOR and SEDA take 500 ms and 1.58 s to recover from congestion collapse, respectively.

1 Introduction

Modern datacenter applications are composed of a set of microservices [15, 16, 36], which use Remote Procedure Calls (RPCs) to interact. To satisfy the low latency requirements of modern applications, microservices often have strict Service Level Objectives (SLOs), some measured in microseconds. Examples of microsecond-scale microservices include services that operate on memory-resident data, such as key-value stores [2, 25] or in-memory databases [41, 47]. Achieving microsecond-scale SLOs is possible under normal loads due to recent advances in operating systems [40] and network hardware [1]. However, maintaining tight SLOs remains a challenge during overload, when the load on a server approaches or exceeds its capacity.

Server overload can cause *receive livelock* [33], where the server builds up a long queue of requests that get starved because the server is busy processing new packet arrivals instead of completing pending requests. This scenario is especially challenging for microsecond-scale RPCs because small delays or bottlenecks can cause SLO violations. Further, the small resource requirements of a short RPC allows a single server to process millions of requests per second, potentially from thousands of clients [10, 35, 50]. Thus, server overload can be caused by "RPC incast" [39, 48], where a large number of clients make requests simultaneously, leading to large queue build-up at the server.

The goal of overload control is to shed excess load to ensure both high server utilization and low latency. Existing overload control schemes broadly fall into two categories. One class of approaches drop requests at an overloaded server or proxy [11, 32, 38]. Other schemes throttle the sending rate of requests at clients [4, 29, 46]. Neither of these approaches performs well for short, microsecond-scale RPCs. Dropping very short requests at the server is not practical as the overhead is comparable to the service time of the request. On the other hand, client-based rate limiting requires clients to know the state of congestion at the server to accurately configure their rate limit, but it takes at least a network round-trip time (RTT) to obtain this information. For requests with service times comparable to the RTT, the delay in reacting to congestion can hurt performance significantly.

A further challenge is to scale the overload control system to large numbers of clients. In a large-scale system, many clients have sporadic demand for a specific server, sending it requests infrequently. Determining the right rate limit for such clients is difficult since they have a stale view of the extent of congestion at the server when making a request. One solution is to explicitly probe the server before sending a request. However, exchanging messages per request to obtain congestion information can impose a high overhead for microsecond-scale RPCs.

In this paper, we present Breakwater, an overload control system for µs-scale RPCs. Breakwater relies on a server-driven admission control scheme where clients are allowed to send requests only when they receive credits from the server. It uses queuing delay at the server as the overload signal. If queuing delay is below an SLO-dependent threshold, Breakwater issues more credits to clients. Otherwise, it reduces the number of credits it issues.

Breakwater minimizes the overhead of coordination (i.e., the communication overhead for the server to know which clients need credits) using *demand speculation*. In particular, a Breakwater server only receives demand information from clients when such information can be piggybacked on requests. When all known demand is satisfied, the server distributes credits randomly to clients. This approach does not require coordination messages to determine demand in clients. However, demand speculation can lead to issuing credits to clients who do not need them at that moment. These unused credits lower server utilization. Thus, Breakwater issues extra credits to ensure high utilization. Such overcommitment introduces the potential for queue buildup at the server if many clients with credits send requests simultaneously (i.e., RPC incast). To mitigate the negative side effects of incast, Breakwater employs delay-based AQM to drop requests that arrive in bursts.

We implemented Breakwater as an RPC library on top of the TCP transport layer. Our extensive evaluation of various workloads demonstrates that Breakwater achieves higher goodput with lower tail latency compared to SEDA [48] and DAGOR [51], the best available overload control systems. For example, Breakwater achieves 6.6% more goodput and 1.9× lower 99%-ile latency with clients' demand of 2× capacity, compared to DAGOR with a synthetic workload. In addition, Breakwater scales to a large number of clients without degrading its benefits. For example, when serving 10,000 clients with memcached, Breakwater achieves 14.3% more goodput and 2.9× lower 99%-ile latency than DAGOR. Compared to SEDA for the same workload, Breakwater achieves 5% more goodput and 1.8× lower 99%-ile when the clients' demand is 2× capacity.

Breakwater is available as open-source software at https://inhocho89.github.io/breakwater/.

2 Motivation and Background

2.1 Problem Definition and Objectives

Overload control is key to ensuring that backend services remain operational even when processing demand exceeds available capacity. Overload was identified as the main cause of cascading failures in large services [11]. Transient overload can occur for a variety of reasons. For example, it may not be cost-effective to provision enough capacity for maximum load [51]. Services can also experience unexpected overload conditions (faulty slow nodes, thermal throttling, hashing hot spots, etc.) despite capacity planning.

Without proper overload control, a system could experience livelock [33], where incoming requests are starved because the server is busy processing interrupts for new packet arrivals, producing no useful work as the majority of requests fail to meet their SLOs. Even when the average of clients' demand is less than the capacity, short-timescale bursty request arrival can degrade latency for short requests. Microsecond-timescale RPCs are much more prone to performance degradation due to short-lived congestion than RPCs with longer service times [45].

RPCs with microsecond-scale execution time are prevalent in modern datacenters. Such RPCs span a variety of operations on data residing in memory or fast storage like M.2 NVMe SSDs (e.g., key-value stores [2, 25] or in-memory databases [41, 47]). The move towards microservice architectures has only increased the prevalence of such RPCs [15, 16, 36]. Further, a single server must process µs-scale requests at very high rates, possibly from thousands of clients [10, 35, 50]. To cope with µs-scale RPCs, an ideal overload control mechanism should provide the following properties:

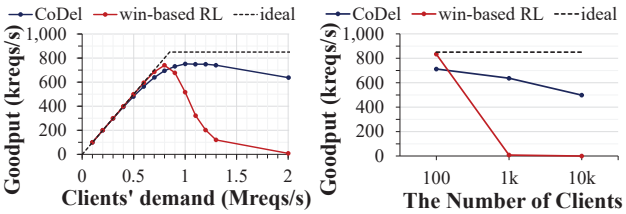
1. No loss in throughput. An RPC server should be processing requests at its full capacity regardless of overload, avoiding livelock scenarios. Further, the overhead of performing the overload control must be minimal.

2. Low latency. An ideal overload control scheme should ensure that any request that gets processed spends minimal time queued at the server. Low queuing latency ensures that processed RPCs meet their SLOs, and is particularly important for µs-scale RPCs which tend to have tight SLOs.

3. Scaling to a large number of clients. For such short RPCs, clients with sporadic demand consume very little resources at the server. Thus, high server utilization requires scaling to a large number of clients. The ideal overload control system should be resilient to "incast" scenarios when a large number of clients send requests within a short period of time. In particular, overload control should prevent queue build-ups that result from incast without harming throughput.

4. Low drop rate. Dropping requests wastes resources at the server because it must spend time processing and parsing packets that will eventually be dropped. Furthermore, dropping requests harms the tail latency of RPCs, especially when network round-trip time (RTT) is comparable to RPC execution time, making retries more expensive. Thus, overload control should minimize the drop rate at the server.

5. Fast feedback. Clients have more flexibility to decide the next action if they can discover when a request is unlikely to be served within its SLO. Thus, if a server expects a request will violate its SLO, it should notify the client as soon as possible so that it can decide an alternative action without having to wait for the request to timeout (e.g., giving up on the request, sending it to another replica, issuing a sim-



(a) Goodput vs. clients' demand with 1,000 clients

(b) Goodput vs. # clients with clients' demand of 2M reqs/s

Figure 1: Goodput of CoDel and window-based rate limiting with different clients' demands and different numbers of clients

pler alternative request, degrading the quality of the service, etc. [18]).

Next, we examine existing overload control mechanisms, which were developed for RPCs with relatively long execution times. Our goal is to understand the challenges of designing an overload control system for µs-scale RPCs.

2.2 Overload Control in Practice

The fundamental concept in overload control is to shed excess load before it consumes any resources [33]. This is typically achieved by either dropping excess load at the server or throttling the sending rate of requests at the client. We look at the performance impairments of these two popular overload control approaches, developed for RPCs with long execution times, when used for µs-scale RPCs.

Active Queue Management (AQM). Such approaches operate as circuit breakers, dropping requests at a server or at a separate proxy under certain conditions of congestion. The simplest approach maintains a specific number of outstanding requests in the queue at the server, typically manually tuned by the server operator [11, 32, 37]. More advanced algorithms can improve performance and avoid the need for manual tuning. For example, CoDel maintains the queuing delay within a specific target value, dropping requests if the queuing delay exceeds the target [11, 32, 38]. RPC servers are typically required to report on success and on failure to avoid expensive timeouts [2, 37, 51]. This means that packets are processed, and failure messages are generated for dropped requests. This overhead is trivial when the message rate is low with a long execution time. However, it becomes a significant overhead in the case of µs-scale RPCs.

To demonstrate the limitations of the AQM approach, we implemented an RPC server that uses CoDel for AQM. Our main evaluation metric is the *goodput* of the server, defined as the throughput of requests whose response time is less than the SLO. Figures 1 (a) and (b) demonstrate the goodput of CoDel with different clients' demands and different numbers of clients. This experiment uses a synthetic workload of requests with exponentially-distributed service time, with a mean of 10 µs. The drop threshold parameter is tuned to achieve the highest goodput given an SLO of 200 µs. As the clients' demand increases, more CPU is used for packet processing even though majority of requests are dropped at server. As a result, less CPU can be used for RPC execution, which leads to goodput degradation. The goodput degradation gets worse with more number of clients. The reason is that the overhead of sending failure messages increases with more clients since fewer messages can be coalesced with the increased number of clients.

Client-side Rate limiting. In order to eliminate the overhead caused by dropping requests at the server, some overload control mechanisms limit the sending rate at the clients. With client-side rate limiting, clients probe the server, detect its capacity, and adjust their rate to avoid overloading the server [4, 29, 46, 49]. The reaction of clients to overload is delayed by a network RTT, which can lead to long delays when the execution time of RPCs is comparable to or less than the RTT. Further, the delay in getting feedback increases with the number of clients; consider the impact this has on overload control performance.

When the number of clients is small, the load generated by each individual client is large and each client exchanges messages with the server at a high frequency. This means that each client has a fresh view of the state of the server, allowing it to react quickly and accurately to overload. In this case, client-based approaches outperform AQM approaches because they have fresh enough information to prevent overload at the server.

As the number of clients increases, the load generated by each client becomes more sporadic and messages are exchanged at a lower frequency between any individual client and the server. This means that in the presence of a large number of clients, each client will have a stale or inaccurate estimate of server overload, leading to clients undershooting or overshooting the available capacity at the server. When many clients overshoot server capacity, it can lead to incast congestion, causing large queueing delays. AQM avoids high tail latency by dropping excess load at the server, leading to AQM outperforming client-based approach for a large number of clients, despite having less than ideal goodput.

To illustrate the limitation of client-side rate limiting with µs-scale execution time, we implement window-based rate limiting used in ORCA [29]. The mechanism is similar to TCP congestion control. The client maintains a window size representing the maximum number of outstanding requests. Upon receiving a response, if the response time is less than the SLO, it additively increases the window size; otherwise, it multiplicatively decreases the window size. Figure 1 (a) and (b) depict the goodput of window-based rate limiting for exponentially-distributed service time of 10 µs (SLO = 200 µs) on average. We optimized the parameters (i.e. additive factor and multiplicative factor) to achieve the highest goodput. Window-based schemes typically support a minimum of one open slot in the window (i.e., a minimum of one outstanding request at the server). This is problematic when there is a large number of clients as each client can

always send one request, leading to incast and overwhelming the server. Rate-based rate limiting [4, 49] overcomes this limitation, but it still suffers from incast with a larger number of clients which results in high latency and low goodput.

Hybrid approaches that combine client-side rate limiting and AQM have also been proposed. We provide a more comprehensive evaluation of rate-based rate limiting and hybrid approaches in §5.

2.3 Challenges

Existing overload control schemes, developed for long RPCs, suffer significant performance degradation when handling µs-scale RPCs. The fundamental challenge facing existing schemes is the need for coordination of clients in order to schedule access to the server under very tight timing constraints. This challenge is exacerbated by the following characteristics of short RPCs:

1. Short average service times. We aim to support execution times for RPCs on the order of microseconds. This requires devising an overload control scheme that can react at microsecond granularity while keeping coordination overheads significantly less than request service times. Achieving this compromise is challenging, and any errors in devising or implementing the overload control scheme can lead to either long queues and overload, or underutilization of the server.

2. Variability in service times. RPC execution times typically follow a long-tailed distribution [11, 17, 18]. The stochastic nature of RPC service times limits the accuracy of any coordination or scheduling at the client or server. Accurate scheduling requires knowledge of the execution time of each request in advance, which is not possible in the presence of long-tailed variability of execution times. Further, this variability creates ambiguity for overload detection because a single request can be long enough to cause significant queueing delay.

3. Variability in demand. Scheduling the access of clients to the server requires some knowledge of both the demand of clients and the capacity of the server. RPCs have various arrival patterns, and clients can have sporadic demand with periods of inactivity [10, 50]. Variability in demand can lead to low utilization because clients that are granted access to server capacity might not have enough demand to utilize it.

4. Large numbers of clients. All previous challenges are exacerbated as the number of clients increases: accurate coordination becomes more challenging and overheads become higher (§5.2). Furthermore, a larger number of clients increases demand variability because it makes the system more susceptible to bursts (i.e., many clients generating demand simultaneously).

The challenges a server overload control system faces bear some similarities to those observed in network congestion control. At a surface level, network and compute congestion can be managed by similar mechanisms, but they each have fundamentally different requirements. Both are necessary to achieve good performance. Network congestion control aims to maintain short packet queues at switches while maximizing network link utilization. By contrast, overload control aims to maintain short request queues at the RPC server while maximizing CPU utilization. There are two critical differences between these problems: (a) RPC processing often has high dispersion in request service times while packet processing times are constant, and (b) client-side demand can fluctuate more significantly at the RPC layer because clients may give up after a timeout or choose to send an RPC to a backup server. On the other hand, once a network flow starts, it generally completes. With such high variability in processing time and demand, designing an overload control system requires overcoming different challenges than a network congestion control system.

2.4 Our Approach

Our work begins with insights from receiver-driven mechanisms proposed in recent work on datacenter congestion control. In receiver-driven congestion control, a receiver issues explicit credits to senders for controlling their packet transmissions, which provides better performance than conventional sender-based schemes [14, 24, 34]. Inspired by this line of work, our design has the following components:

1. Explicit server-based admission control: A client is only allowed to send a request if it receives explicit permission from the server. A server-based scheme allows for coordination that is based on the accurate estimation of the state of the server. Explicit admission control means that the load received by the server is completely controlled by the server itself. This allows for more accurate control that maintains high utilization and low latency. Server-based admission control can add an extra RTT for a client to request admission. We avoid this through piggybacking and overcommitting credits, as detailed later.

2. Demand speculation with overcommitment: The server requires knowledge of clients' demand in order to decide which client should be permitted to send requests. This is comparable to the need for clients to know about the state of the server in client-based schemes. Exchanging such information introduces significant overhead as the number of clients increases. Furthermore, as the execution time of RPCs decreases, the frequency of exchanging the demand information increases, further increasing overhead. The key difference between server-based schemes and client-based schemes is that we can relax the need for the server to have full information about clients' demand without harming performance. In particular, we allow the server to speculate about clients' demand and avoid lowering server utilization by allowing the server to overcommit, issuing more credits than its capacity.

3. AQM: Due to overcommitment, the server can occasionally receive more load than its capacity. Thus, we rely on AQM to shed the excess load. In our scheme, the need for AQM to drop requests is rare, as credits are only issued when the server is not overloaded.

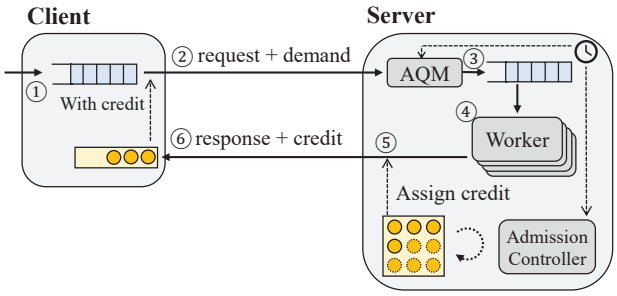


Figure 2: Breakwater overview

3 System Design

We present Breakwater, a scalable overload control system for µs-scale RPCs. Figure 2 depicts an overview of the interaction between a Breakwater client and server pair. A new client joining the system sends a register message to the server, indicating the number of requests it has in its queue. The client piggybacks its first request to the registration message. The server adds the client to its client list, and if it is not overloaded it executes the request. The server then replies to the client with the execution result or a failure message. The server piggybacks with the response any credits it issued to the client depending on the demand indicated by the client. The client issues more requests depending on the number of credits it received. When the client has no further requests, it sends a deregister message to the server returning back any unused credits.

For the rest of the section, we present how Breakwater detects overload and how it reacts to it. In particular, we present how a server determines the number of credits it can issue, how to distribute them among clients, and how clients react to credits or the lack thereof.

3.1 Overload Detection

There are multiple signals we can utilize to determine whether a server is congested. CPU load is a popular congestion signal—it is often used to make auto-scaling decisions in cloud computing [5]. However, CPU utilization indicates only one type of resource contention that can affect RPC latency. For instance, requests contending for a hard disk can have high latency, but CPU utilization will remain low [22]. Moreover, using CPU utilization as a signal does not allow an overload controller to differentiate between the ideal scenario of 100% utilization with no delayed RPCs and a livelock state.

Another potential congestion signal is queue length at the server. A similar signal is widely used in network congestion control [8, 52]. Unfortunately, when RPC service times have high dispersion, queue length is a poor indicator of request latency. A more reliable signal is queuing delay, as it is accurate even under RPC service time variability. Furthermore, it is intuitive to map a target SLO to a target queueing delay at the server. Thus, Breakwater uses queuing delay as its congestion signal.

Effective overload control requires accurate measurement of the queuing delay signal. In particular, the signal should account for the sum of each of the queueing delay stages a request experiences, ignoring non-overload induced delays. This ensures that the system only curbs incoming requests when it is overloaded. This is especially critical for microsecond-scale RPCs, as they leave little room for error.

Breakwater has two stages of queueing. Packets are queued while they await processing to create a request. Then, threads created to process requests are queued awaiting execution. Breakwater tracks and sums queuing delay at both of these stages. In particular, for every queue in the system, each item (e.g., a packet or a thread) is timestamped when it is enqueued. Each queue maintains the oldest timestamp of enqueued elements in a shared memory region, and this timestamp is updated on every dequeue. When the delay of a queue needs to be calculated, Breakwater computes it by taking the difference between the current time and the queue's oldest timestamp. We use this approach instead of measuring explicit delays of each request (i.e., the timestamp difference between request arrival and the request execution) because we must keep track of the total queueing delay as a request moves from one queueing stage to another.

There are multiple sources of delay that are not caused by high utilization or overload. For example, long delays due to head-of-line blocking do not indicate a thread is waiting for resources, but rather it is a sign of poor load balancing. Accurate queueing delay measurement requires the system to avoid such delays. We find that the biggest source of such delays is the threading model used by the system. Our initial approach for developing Breakwater relied on the in-line threading model [19, 25] where a single thread handles both packet processing and request processing. This choice was made as the in-line model provides the lowest CPU cost. However, it leads to head-of-line blocking as a single request with a large execution time can block other requests waiting at the same core. The alternative is relying on the dispatcher threading model [41] where a dispatcher thread processes packets and spawns a new thread for request processing incurring inter-thread communication overhead. However, this overhead is minimal when the dispatcher model is implemented using lightweight threads in recently proposed low-latency stacks (e.g., Shenango [40] and Arachne [43]). Thus, Breakwater employs the dispatcher model for request processing.

3.2 Overload Control

During overload, the system has to decide which requests to admit for processing and which requests to drop or possibly queue at the client. In this section, we explain our design for Breakwater's approach to overload control.

3.2.1 Server-driven Credit-based Admission Control

A Breakwater server controls the admission of incoming requests through a credit-based scheme. Server-driven admission control avoids the need for clients to probe the server to know what rate to send at. It also allows the server to receive the exact load it can handle. A credit represents availability

at the server to process a single request by the client that receives the credit. A Breakwater server manages a global pool of credits (C_{total}) that is then distributed to individual clients. C_{total} represents the load the server can handle while maintaining its SLO. This is achieved by controlling C_{total} such that the measured queuing delay (d_m) remains close to a target queuing delay (d_t), which is set based on the SLO of the RPC.

Every network RTT, Breakwater updates C_{total} based on the measured queuing delay (d_m). If d_m is less than d_t, Breakwater increases C_{total} additively.

$$C_{total} \leftarrow C_{total} + \mathcal{A} \tag{1}$$

Otherwise, it decreases C_{total} multiplicatively, proportional to the level of overload.

$$C_{total} \leftarrow C_{total} \cdot \max(1.0 - \beta \cdot \frac{d_m - d_t}{d_t}, 0.5) \tag{2}$$

Note that $\mathcal{A}$ controls the overcommitment and aggressiveness of the generation of credits. On the other hand, β controls the sensitivity of Breakwater to queue build-up. We explain how we select $\mathcal{A}$ and β in the next section.

Once C_{total} is decided, credits are distributed to clients. When C_{total} increases, new credits are issued to clients by piggybacking the issued credits to response messages sent to the clients. Explicit *credit* messages are only generated when piggybacking is not possible (i.e., server has no messages bound for the client). When C_{total} decreases, the server does not issue additional credits to the clients, or if the clients have unused credits, the server sends negative credits to revoke the credits issued earlier. The server can tell how many unused credits each client has by keeping track of the number of credits issued and the number of requests received. In the following section, we explain how Breakwater decides which client should be issued credits.

3.2.2 Demand Speculation with Overcommitment

There is a tradeoff between accurate credit generation and messaging overhead. Choosing which client should receive a credit can be simply determined based on the demand at the client. This requires clients to inform the server whenever their number of pending requests changes. The server can then select which clients to send a credit to based on demand. This ensures that all issued credits are used, allowing the server to generate credits that accurately represent its capacity. However, as we scale the number of clients, the overhead of exchanging demand messages overwhelms the capacity of the server.

In our design of Breakwater, we choose to eliminate the messaging overhead completely. A client notifies the server of its demand only if the demand information can be piggybacked on a request (i.e., the client already has a credit and can send a request to the server). The server therefore does not have accurate information about clients with sporadic demand as they can't update the server as soon as their demand changes. Thus, Breakwater speculatively issues credits based on the latest demand information even though it may be stale. Speculative generation of credits means that some clients that receive credits will not be able to use them immediately. If credits are generated to exactly match capacity, the server may experience underutilization because some credits are left unused when they are issued to clients with no queued requests. To achieve high utilization, speculative demand estimation is coupled with credit overcommitment to ensure that enough clients receive credits to keep the server utilized.

Overcommitment is achieved by setting the $\mathcal{A}$ and β parameters of the admission control algorithm. In particular, we set $\mathcal{A}$ to be proportional to the number of clients (n_c).

$$\mathcal{A} = \max(\alpha \cdot n_c, 1) \tag{3}$$

where α controls the aggressiveness of the algorithm. Further, each client is allowed to have more credits than its latest demand. The number of overcommitted credits per client (C_{oc}) is based on the number of clients (n_c), the total number of credits in the credit pool (C_{total}), and the total number of credits presently issued to clients (C_{issued}).

$$C_{oc} = \max(\frac{C_{total} - C_{issued}}{n_c}, 1) \tag{4}$$

The server makes sure that each client does not have unused credits more than its (latest) demand plus C_{oc} by revoking already issued credits if necessary.

Further, Breakwater attempts to avoid generating explicit credit messages whenever possible. This means that a new credit will be given to a client to whom the server is about to send a response unless that client has reached the maximum number of credits it can receive. Explicit credit messages are only generated when piggybacking a credit on a response is not possible. In the current version of Breakwater, the client that receives an explicit credit message is selected randomly, but we expect the selection could be smarter with per-client statistics. For example, the server can choose a client based on its average request rate to increase the likelihood of the client using the credit immediately.

3.2.3 AQM

The drawback of credit overcommitment is that the server may occasionally receive a higher load than its capacity, leading to long queues. To ensure low tail latency at all times, Breakwater relies on delay-based AQM to drop requests if the queueing delay exceeds an SLO-derived threshold. In our results, we find that drops are rare because our credit-based admission control scheme avoids creating bursts. Drops can be further reduced with by setting a large SLO budget. In particular, a system administrator can set a large threshold for AQM to reduce the drop rate at the expense of having a looser SLO.

3.3 Breakwater Client

Breakwater allows a client to queue requests if it does not have a credit for it. Client-side queuing is critical in a server-driven system as the client has to wait for the server to admit a request before it can send it. However, if the client queue is too long, the request will experience high end-to-end latency. In Breakwater, in order to achieve high throughput and low end-to-end latency, we allow requests to expire at the client. The request expiration time is set based on its SLO.

When a client receives credits, it can immediately consume them if its queue length is equal to or larger than the number of credits it receives. Due to overcommitment, a client can receive credits which it cannot immediately consume (c_{unused}). When a client receives negative credits with decreased C_{total} at the server, the client decrements c_{unused}. However, if a client has already consumed all of its credits (i.e., $c_{unused} = 0$), no action is taken by the client.

4 Implementation

Breakwater requires a low-latency network stack in order to ensure accurate estimation of the queuing delay signal. This requires minimal variability in packet processing and no head-of-line-blocking between competing requests. We use Shenango [40], an operating system designed to provide low tail latency for µs-scale applications with fast core allocations, lightweight user-level threads, and an efficient network stack. Shenango achieves low latency by dedicating a busy-spinning core to reallocate cores between applications every 5 µs to achieve high utilization and minimize the latency of packets arriving into the server.

We implement Breakwater as an RPC library on top of the TCP transport layer. Breakwater handles TCP connection management, admission control with credits, and AQM at the RPC layer. Breakwater abstracts connections and provides a simple individual RPC-oriented interface to applications, leaving applications to only specify request processing logic. Breakwater provides a single RPC layer per application (i.e., overload signal, credit pool, etc.) regardless of the number of cores allocated to the application and the number of clients of that application. A request arriving at a Shenango server is first queued in a packet queue. Then a Shenango kernel thread processes packets and moves the payload to the socket memory buffer of the connection. Once all the payload of a request is prepared in the memory buffer, a thread in Breakwater parses the payload to a request and creates a thread to process it. Threads are queued pending execution, and when they execute, they execute to completion.

Threading model. As explained earlier, Breakwater relies on a dispatcher threading model for accurate queueing delay measurement. A Breakwater server has a listener thread and the admission controller thread running. When a new connection arrives, the listener thread spawns a receiver thread and a sender thread per connection. Receiver threads read incoming packets and parse them to create requests. After parsing a request, AQM is performed, dropping requests if the current queueing delay is greater than the AQM drop threshold. If a request is not dropped, the receiver thread spawns a new thread for the request. The new thread is enqueued to the thread queue. The sender thread is responsible for sending responses (either success or reject) back to the clients. If there are multiple responses, the sender thread coalesces them to reduce the messaging overhead. For all threads in Breakwater, we use lightweight threads provided by Shenango's runtime library.

Queueing Delay Measurement. With a separate receiver thread minimizing the delay from the socket memory buffer, the two main sources of queueing delay in Shenango are packet queueing delay (i.e., time between when a packet arrives till it is processed by a Shenango kernel thread) and thread queueing delay (i.e., time between when a thread is created to process a request until it starts executing). In Shenango, each core has a packet queue and a thread queue shared with IOKernel. We instrumented packet queues and thread queues so that each queue maintains the timestamp of the oldest item, and we modified Shenango's runtime library to export the queueing delay signal to the RPC layer. When Shenango's runtime is asked for the queueing delay, it returns the maximum of the packet queue's delays plus the maximum of the thread queue's delays.

Lazy credit distribution. The admission controller updates C_{total} every RTT. Once the credit pool size is updated, the admission controller can re-distribute credits to clients to achieve max-min fairness based on the latest demand information. However, this requires the admission controller to scan the demand information of all clients, requiring $O(N)$ steps. To reduce the credit distribution overhead, Breakwater approximates max-min fair allocation with lazy credit distribution. In particular, Breakwater delays determining the number of credits a client can receive until it has a response to send to that client. The sender thread, responsible for sending responses to a client, decides whether to issue new credits, not to issue any credits, or to revoke credits based on C_{issued}, C_{total}, and the latest demand information. It first calculates the total number of credits the server should grant to client x (c_x^{new}). If C_{issued} is less than C_{total}, c_x^{new} becomes

$$c_x^{new} = \min(demand_x + C_{oc}, c_x + C_{avail}) \quad (5)$$

where $demand_x$ is the latest demand of client x, c_x is the number of unused credits already issued to client x and C_{avail} is the number of available credits the server can issue ($C_{avail} = C_{total} - C_{issued}$). If C_{issued} is greater than C_{total}, c_x^{new} becomes

$$c_x^{new} = \min(demand_x + C_{oc}, c_x - 1) \quad (6)$$

The sender thread then piggybacks the number of credits newly issued for client x ($c_x^{new} - c_x$) to the response. It also updates c_x to c_x^{new} and C_{issued} accordingly.

5 Evaluation

Our evaluation answers the following questions:

- Does Breakwater achieve the objectives of overload control defined in §2 even given tight SLOs?
- Can Breakwater maintain its advantages regardless of load characteristics (i.e., average RPC service time and service time distribution)?
- Can Breakwater scale to large numbers of clients?
- Can Breakwater react quickly to a sudden load shift?
- What is the impact of Breakwater's key design decisions: demand speculation and credit overcommitment?
- How sensitive is Breakwater's performance to different parameters?

5.1 Evaluation Setup

Testbed: We use 11 nodes from the Cloudlab xl170 cluster [20]. Each node has a ten-core (20 hyper-thread) Intel E5-2640v4 2.4 GHz CPU, 64 GB ECC RAM, and a Mellanox ConnectX-4 25 Gbps NIC. Nodes are connected through a Mellanox 2410 25 Gbps switch. The RTT between any two nodes is 10 µs. We use one node as the server and ten nodes as clients. The server application uses up to 10 hyper-threads (5 physical cores) for processing requests, and the client application uses up to 16 hyper-threads (8 physical cores) to generate load. All nodes dedicate a hyper-thread pair for Shenango's IOKernel.

Baseline. We compare Breakwater to DAGOR [51] and SEDA [48]. DAGOR is a priority-based overload control system used for WeChat microservices. Priorities are assigned based on business requirements across applications and at random across clients. We only consider a single application in our evaluation. DAGOR uses queueing delay to adjust the priority threshold at which a server drops incoming requests (i.e., requests with a priority lower than the threshold are dropped). To reduce the overhead of dropped requests, the server advertises its current threshold to clients, piggybacked it in responses. Clients use that threshold to drop the requests. Note that DAGOR does not drop its threshold to zero, meaning that a request with the highest priority value (i.e., a priority of one) will never be dropped. SEDA uses a rate-based rate limiting algorithm. It sets rates based on the 90%-ile response time. Since we evaluate the performance of Breakwater using the 99%-ile latency metric, we modified SEDA's algorithm so that it adjusts rates based on 99%-ile response time. We implement DAGOR and SEDA as an RPC layer in Shenango with the same dispatcher model as Breakwater.

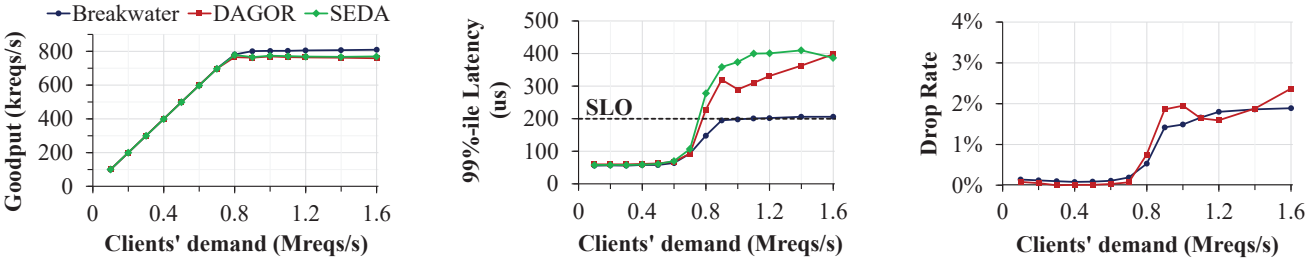
Setting end-to-end SLO. We set tight SLOs to support low-latency RPC applications. We budget SLOs based on the server-side request processing time and the network RTT. An SLO is set as 10× the sum of the average RPC service time measured at the server and the network RTT; the multiplicative factor of 10 was inspired by recent work on µs-scale RPC work [17, 42]. The RTT in our setting is 10 µs, leading to SLOs of 110 µs, 200 µs, and 1.1 ms for workloads with 1 µs, 10 µs, and 100 µs average service times, respectively. These are comparable with SLO values used in practice [30].

Evaluation metrics: We report goodput, 99%-ile latency, drop rate, and reject message delay. Goodput represents the number of requests processed per second that meet their SLO. Reported latency captures all delays faced by a request from the moment it is issued till its response is received by the client. This includes any queuing delay at the client, communication delay, and all delays at the server. We report drop rate at the server only, as it is the factor the directly impact overall system performance. *Note that SEDA does not support any AQM at the server and has zero drop rate in all experiments.* Reject message delay represents the delay between the departure of a request from a client and the arrival of a reject message back to the client when that request is dropped at the server.

Parameter tuning. We tune the parameters of all systems so that they achieve the highest possible goodput. We re-tune the parameters when we change the average service time, service time distribution, and the number of clients. Note that Breakwater and DAGOR do not require parameter re-tuning for a different number of clients while SEDA does. Specifically, we need to scale adj_i parameter in SEDA based on the number of clients to get the best goodput. For Breakwater, we set $\alpha = 0.1\%$, $\beta = 2\%$, d_t to 40% of SLO, and AQM threshold to $2 \cdot d_t$ (e.g., $d_t = 80$ µs and AQM threshold = 160 µs for exponential service time distribution with 10 µs average and 200 µs SLO). For DAGOR and SEDA, which are devised for ms-scale RPCs, we scale down the hyperparameters from the default values. For DAGOR, we update the priority threshold every 1 ms (instead of 1 s) or every 2,000 requests and use $\alpha = 5\%$ and $\beta = 1\%$. We assign random priority for each request ranging from 1 to 128, which is the default priority setting with one type of service in DAGOR [51]. We tune $DAGOR_q$ for each workload (e.g., $DAGOR_q = 70$ µs for exponential service time distribution with 10 µs on average). For SEDA, we used the same default parameter from [48] except for $timeout$, adj_i, and adj_d. We set $timeout = 1$ ms (instead of 1 s) and tune adj_i and adj_j for each workload (e.g., $adj_i = 40$, $adj_d = 1.04$ for exponential workload with 10 µs average with 1,000 clients). AQM in Breakwater and DAGOR drops requests right after parsing packets to requests, following the drop-as-early-as-possible principle [33]. We run all the experiments for four seconds. We measure steady state performance with converged adaptive parameters by collecting data two seconds after an experiment starts.

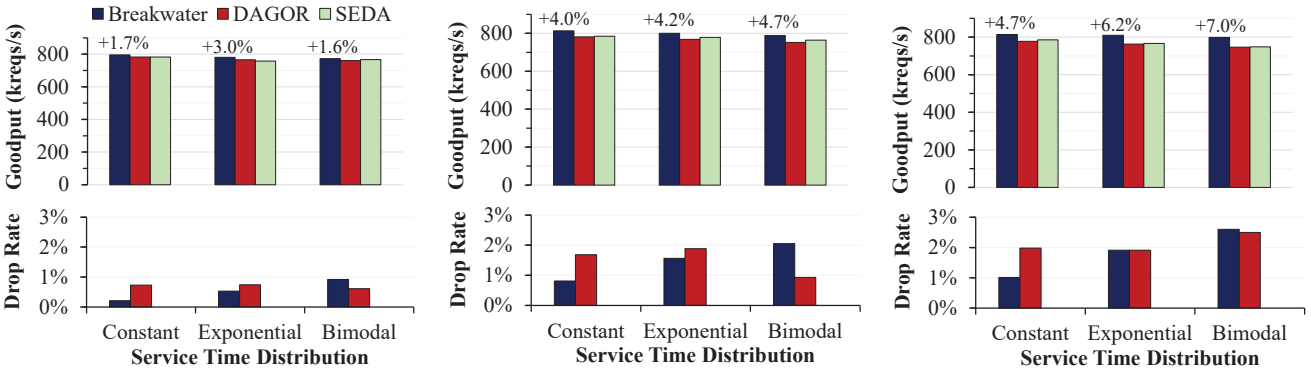
5.2 Performance for Synthetic Workload

Workload: We run 1,000 clients divided equally between the ten nodes in our CloudLab setup. We generate the workload with exponential, constant, and bimodal service time distributions with 1 µs, 10 µs, and 100 µs average where each client generates the load with an open-loop Poisson process. We change the demand by varying the average arrival rate of requests at the server between 0.1× to 2× of server capac-



(a) Goodput (b) 99%-ile Latency (c) Drop Rate

Figure 3: Performance of Breakwater, DAGOR, and SEDA for synthetic workloads with the exponential distribution of 10µs average



(a) demand = 0.9× capacity (b) demand = 1.2× capacity (c) demand = 2× capacity

Figure 4: Goodput and drop rate with different service time distribution of 10µs average with 1,000 clients (The label represents the goodput gain compared to the worst of baselines.)

ity. Exponential service time distribution models applications waiting for a shared resource while busy-spinning; constant distribution models applications with a fixed amount of latency such as fetching value from memory or flash drive; bimodal distribution models applications that caches frequently requested values, which will have shorter execution time compared non-cached results. In particular, 20% of the requests take four times the average service time, and 80% of the requests take one fourth of the average following the Pareto principle.

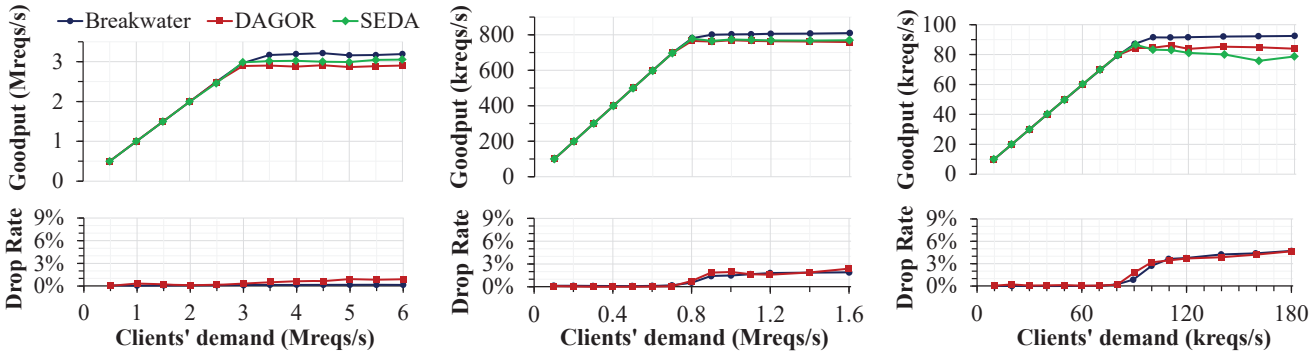
Overall performance: Figure 3 shows the performance for a workload whose service time follows an exponential distribution with 10µs average. The capacity of the server in this case is around 850k requests per second.

When the clients' demand is less than the capacity, all three systems perform comparably in terms of goodput, latency, and drop rate. The only noticeable difference among them is that, at 700k reqs/s, SEDA has a 15% higher 99%-ile latency than Breakwater or DAGOR. This is because SEDA doesn't drop requests at servers.

When the clients' demand is around the capacity of the server, Breakwater achieves 801k requests per second for goodput (or 808k reqs/s of throughput), which is around 5% overhead when compared to the maximum throughput with no overload control. Other systems have higher overhead than Breakwater.

When the demand exceeds the capacity, incast becomes the dominant factor impacting performance. Breakwater handles incast well by preventing clients from sending requests unless they have credits, limiting the maximum queue size. Thus, Breakwater achieves higher goodput with lower and bounded tail latency. On the other hand, SEDA experiences high tail latency because clients do not coordinate their rate increase, making multiple clients increase their rate simultaneously and overwhelm the server. Delayed reaction to overload does not allow SEDA to react quickly to incast. DAGOR's high tail latency is also explained by delayed reaction as it updates its priority threshold every 1 ms or every 2,000 requests. Breakwater is also impacted by incast due to the overcommitted credits, which lead to increased tail latency and higher drop rate with overload. However, Breakwater relies on delay-based AQM which effectively bounds the tail latency while maintaining a comparable drop rate to DAGOR.

Impact of Workload Characteristics: To verify that Breakwater's performance benefits are not confined to a specific workload, we repeat the experiments with different service time distributions and different average service time values. Figure 4 shows goodput and drop rate with three different distributions of the service time whose average is 10µs, where the load generated by 1,000 clients is 0.9×capacity, 1.2× capacity, and 2× capacity. The service time distributions are aligned over the x-axis in ascending order of variance. Break-



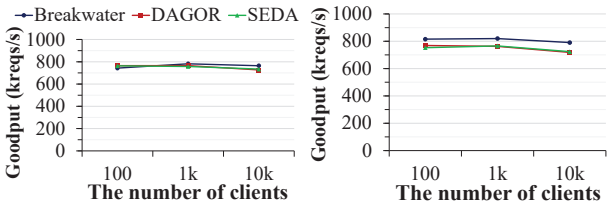
(a) 1 µs (b) 10 µs (c) 100 µs

Figure 5: Goodput and drop rate with different average service time with 1,000 clients

water achieves the highest goodput regardless of the load and service time distribution. All three systems experience small goodput reduction with a higher variance, especially when the load is 2× the server capacity. The goodput reduction of DAGOR and SEDA comes from their poor reaction to incast, whose size increases as the load increases. As a result, Breakwater's goodput benefit becomes larger as the clients' demand increases. Breakwater achieves 5.7% more goodput compared to SEDA and 6.2% more goodput compared to DAGOR with exponential distribution at a load of 2× capacity. With a higher variance of the service time distribution, the drop rate of the Breakwater tends to increase because a larger number of credits are overcommitted with higher variance, but it is still comparable to DAGOR.

Figure 5 depicts performance with an exponential service time distribution and different average service times with 1,000 clients. Breakwater outperforms DAGOR and SEDA regardless of the clients' demand and the average service time. As the average service time increases, clients and servers exchange messages less frequently, exposing the delayed reaction problem in SEDA and DAGOR. With short service times (i.e., 1 µs), clients and servers exchange messages very frequently, giving clients a fresh view of the state of the server in case of DAGOR and SEDA, allowing clients to react quickly to overload. With high demand, the size of incast gets larger which is poorly handled by SEDA and DAGOR. With clients' demand of 2× capacity with 100 µs (i.e., 180k reqs/s), Breakwater achieves 17.5% more goodput than SEDA and 10.2% more goodput with a comparable drop rate compared to DAGOR.

Scalability to a Large Number of Clients: We vary the number of clients from 100 to 10,000 with synthetic workload whose service time follows exponential service time distribution of 10 µs average. Note that the server capacity is around 850k requests per second. Figure 6 depicts the goodput with different numbers of clients. As clients' demand nears and exceeds the capacity, the goodput of all systems degrades as the number of clients increases. As the number of clients increases, the size of incast increases, leading to



(a) demand = capacity (b) demand = 2× capacity

Figure 6: Goodput with different numbers of clients for exponential workload with 10 µs average service time

performance degradation. This is problematic for Breakwater as well since overcommitment can occasionally result in large bursts of incoming requests. The performance of DAGOR and SEDA drops more than Breakwater as the number of clients increases. This is because each client exchanges messages with the server less frequently as the number of clients increases. The stale view of the server status leads clients to overwhelm the server. Note that for SEDA's best performance, we scale the additive rate increase factor ($ad\,j_i$) to the number of clients. This helps mitigate any bursty behavior that can result from multiple clients sharply increasing their rate simultaneously. A small increase factor is not practical for a small number of clients as it will lead to slow ramp-up of rates after an overload, leading to lower utilization of the server. Because of this issue, SEDA has a much slower convergence time to the right rate, making it impractical for load shift scenarios as we show next.

Further, it is hard to tune SEDA dynamically. The rate control algorithm in SEDA is implemented at the client, and dynamic tuning requires each client to know the total number of *active* clients. Such a dynamic approach will lead to performance degradation as the client will retune its parameter to at least an RTT after the number of clients changes. The drawbacks of such a delayed reaction can be seen in the behavior of DAGOR. Further, exchanging such information might not feasible in practice due to messaging overhead as well as privacy concerns (e.g., a FaaS cloud provider will not want any of its clients to know the total number of clients). Note that even though Breakwater also scales the number

of newly issued credits to the number of clients (Equation 1 and 3), Breakwater is server-driven, and the server has perfect knowledge of the number of active clients at all times with no need to expose this information outside. In SEDA, by contrast, each client cannot have perfect knowledge of the number of active clients. Each client would have to guess or receive feedback from the server to scale the increment factor.

Reaction to Sudden Shifts in Demand: An RPC server may experience sudden shifts in demand for many reasons, such as load imbalance, packet bursts, unexpected user traffic, or redirected traffic due to server failure. To verify Breakwater's ability to converge after a shift in demand, we measure its performance with a shifting load pattern. We use a workload whose service time follows an exponential distribution with 10 µs average and calculate goodput, 99%-ile latency, and mean reject message delay every 20 ms. When the experiment starts, 1,000 clients generate requests at 400k reqs/s (0.5× capacity). Then, clients double their request rate to 800k reqs/s (0.9× capacity) at time = 2 s, then triple their demand to 1.2M reqs/s (1.4× capacity) at time = 4 s. Clients sustain their demand at 1.2M reqs/s for 2 seconds. Then, clients reduce their demand back to 800k reqs/s at time = 6 s and finally to 400k reqs/s at time = 8 s. Figure 7 depicts a time series behavior of all systems.

When the clients' demand is far less than the capacity, all three overload control schemes maintain comparable goodput and tail latency at a steady state. When demand increases to near server capacity, Breakwater converges fast, exhibiting a stable behavior in terms of both goodput and tail latency. On the other hand, DAGOR and SEDA experience higher tail latency because of the poor reaction to the transient server overload. As the server becomes persistently overloaded with a sudden spike at time = 4 s, Breakwater converges quickly while DAGOR and SEDA suffer from congestion collapse. Breakwater experiences a momentary tail latency increase (reaching 1.4× the SLO) with the sudden increase of clients' demand due to more incast caused by overcommitted credits. However, credit revocation and AQM rapidly limit the impact of any further incast. When demand returns back below the capacity at time = 6 s, Breakwater doesn't show a noticeable goodput drop while the DAGOR and SEDA experience a temporary goodput drop down to 77.5% and 82.6% of the converged goodput, respectively.

SEDA reacts slowly to the demand spike since each client needs to wait for a hundred responses or 1 ms to adjust its rate. After the demand spikes beyond the capacity, the server builds up long queues, and the latency goes up beyond SLO, resulting in almost zero goodput. SEDA takes around 1.6 s to recover its goodput. DAGOR also has the delayed reaction problem, but its goodput converges more quickly than SEDA thanks to AQM, taking 500 ms to recover its goodput. During the congestion collapse period, the 99%-ile latency of DAGOR soars up to 300 ms and its mean delay of reject message reaches 220 ms. This is problematic as clients cannot receive the feedback in a timely manner, making them rely on expensive timeout.

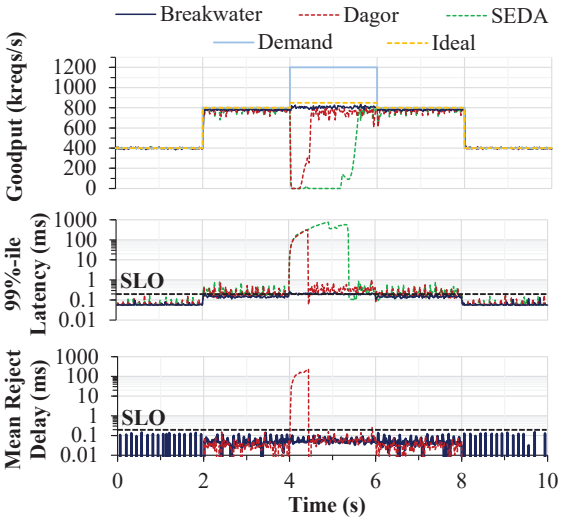


Figure 7: Goodput, 99%-ile latency, and mean rejection delay with a sudden shift in demand with 1,000 clients

The Value of Demand Speculation: To quantify the performance benefits of demand speculation, we compare the two strategies for collecting demand information: demand synchronization and demand speculation. With demand synchronization, clients notify the server whenever their demand changes using explicit demand messages, and the server generates explicit credit messages to clients if it cannot be piggybacked to responses. With demand speculation, the server speculatively estimates client demands based on the latest demand information and piggybacks credits to the responses as much as possible. The load is generated by 1,000 clients where the service time per request follows an exponential distribution with an average of 10 µs. The message overhead is measured by the number of packets received (RX) and sent (TX) at the server. With demand synchronization, both RX and TX message overhead increase as the clients' demand increases, leading to goodput degradation (Figure 8 (a)). In particular, explicit demand and credit messages doubles RX and TX message overhead below and at the capacity (i.e., 850k requests per second). As the system gets overloaded, the overhead of demand messages keeps increasing because per-client demand changes more frequently with increased clients' demand. Further, the overhead of generating credits contributes to the cost of synchronization. The server sends more credit messages during low demand as they cannot be piggybacked on responses due to low request rates. As load increases beyond capacity, more credits can be piggybacked to the responses, which results in the reduction of TX overhead. Demand synchronization has a smaller number of overcommitted credits, leading to a lower drop rate than demand speculation (Figure 8 (c)). Overall, the cost of synchronization between the clients and the server is high in terms of goodput degradation and network overhead, with the small

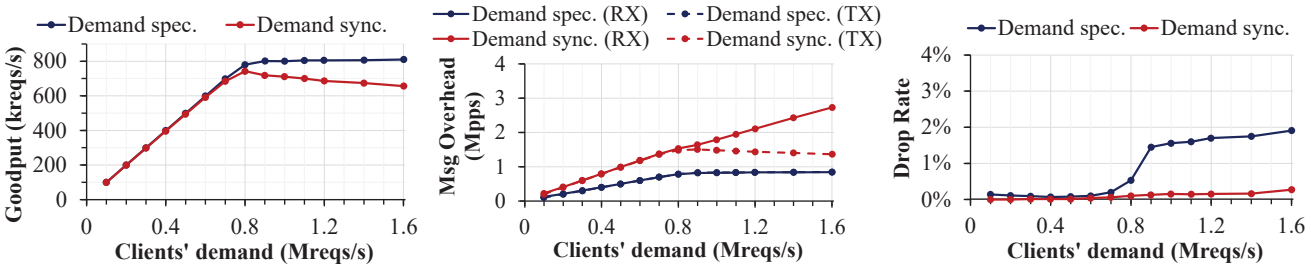


Figure 8: Goodput, message overhead, and drop rate with demand speculation and demand synchronization in Breakwater

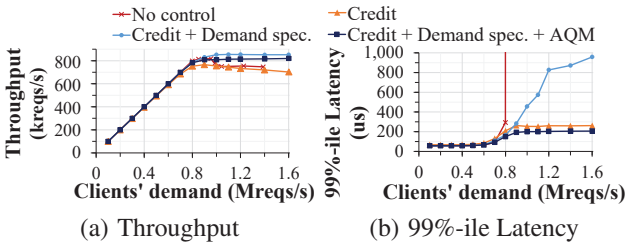


Figure 9: Breakwater performance breakdown

benefit of lowering the drop rate at the server.

Performance Breakdown: To quantify the contribution of each component of Breakwater to its overall performance, we measure the throughput and 99%-ile latency after incrementally activating its three major components: credit-based admission control, demand speculation, and delay-based AQM. The results are shown in Figure 9. We use the synthetic workload whose service time is exponentially distributed with 10 µs average (SLO = 200 µs). With no overload control at all, throughput starts to degrade, and tail latency soars, making almost all requests violate their SLO as demand becomes higher than server capacity. Credit-based admission control effectively lowers and bounds the tail latency, but throughput still suffers due to the messaging overhead. Demand speculation with message piggybacking reduces the messaging overhead, but it worsens tail latency due to incast caused by credit overcommitment. By employing delay-based AQM, Breakwater effectively handles incast, leading to high throughput and low tail latency.

Parameter Sensitivity: Breakwater parameters are set aggressively to maximize the goodput, resulting in a relatively high drop rate. With less aggressive parameters, Breakwater can drop fewer requests sacrificing goodput. Figure 10 demonstrates the trade-off between the goodput and the drop rate for the workload with exponential service time distribution with 10 µs average with 1M reqs/s demand from 1,000 clients. The values of pairs of α and β are aligned in descending order of aggressiveness over the x-axis. Breakwater achieves 0.7% of drop rate by sacrificing 2.2% of goodput (with $\alpha = 0.1\%$, $\beta = 8\%$) and 0.4% of drop rate by sacrificing 5.1% of goodput (with $\alpha = 0.05\%$, $\beta = 10\%$).

In practice, it is not easy to find the best parameter con-

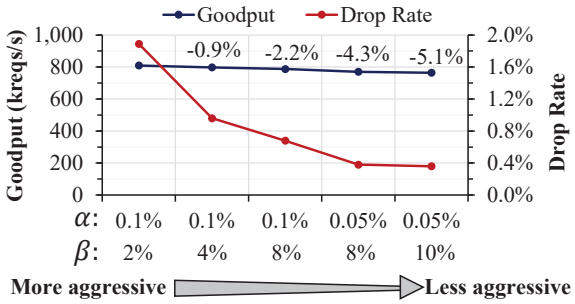


Figure 10: Goodput and drop rate with different aggressive parameters of Breakwater

figuration for an operational system. It is even more difficult when traffic patterns change over time because parameter adjustments could be required to achieve the best possible performance. Thus, it is desirable to develop systems that are robust to parameter misconfiguration and changes in traffic patterns, providing consistently good performance even with small errors in parameter settings. Breakwater is robust. In particular, it provides high throughput and low tail latency despite parameter misconfiguration. We compare it against DAGOR and SEDA, measuring their performance for the same workload while varying their parameters. Specifically, we measure the throughput and 99%-ile latency after reconfiguring the three most sensitive parameters for each system: target delay, increment factor, and decrement factor (d_t, α, β for Breakwater; threshold of the average queueing time, α, β for DAGOR; and $target, ad\,j_i, ad\,j_d$ for SEDA). Given the set of parameters producing best goodput, we measure 27 data points with -10, 0, +10 µs of target queueing delay, 0.5×, 1×, 2× of the increment factor, and 0.5×, 1×, 2× of the decrement factor. We use a synthetic workload with exponentially distributed service times with 10 µs average with 1,000 clients. The results are shown in Figure 11 where the circles filled with light color indicate the performance with the parameters tuned for the best goodput. All configurations of Breakwater achieve comparable performance in terms of both throughput and tail latency, achieving better throughput and latency trade-offs and more consistent performance with different sets of parameters. DAGOR tends to provide high throughput, but its tail latency is as high as four times the SLO in the worst case. SEDA's worst case tail latency is lower than DAGOR,

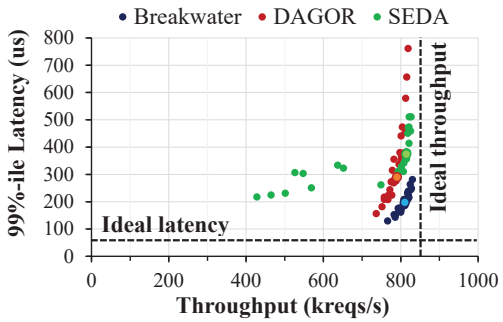


Figure 11: Throughput and 99%-ile latency trade-off with different sets of parameters (circle with light color indicates the point producing best goodput)

but it suffers from severe throughput degradation when its parameters are too conservative.

5.3 Performance under Realistic Workload

To evaluate Breakwater in a more realistic scenario, we create a scenario where one memcached instance serves 10,000 clients. We use the USR workload from [9] where 99.8% of the requests are GET, and other 0.2% are SET. Each client generates the load according to an open-loop Poisson process. We set an SLO of 50 µs considering that the latency of GET operation of memcached is less than 1 µs. Figure 12 shows goodput, median latency, 99%-ile latency, and drop rate of Breakwater, DAGOR, and SEDA. Breakwater achieves steady goodput, low latency, and low drop rate, whereas both DAGOR and SEDA suffer from goodput degradation with high tail latency caused by incast when the server becomes overloaded. With clients' demand of 2× capacity, Breakwater achieves 5% more goodput and 1.8× lower 99%-ile latency than SEDA; and 14.3% more goodput and 2.9× lower 99%-ile latency than DAGOR. Because of bimodally distributed service time with a mix of GET and SET requests, Breakwater shows around 25 µs higher 99%-ile latency than its SLO and about 1.5% point higher drop rate than DAGOR.

6 Discussion and Future Work

Auto-scaling. We do not consider auto-scaling [5, 23, 31] in this paper, where more resources are provisioned as load increases, as a potential solution for overload control. Auto-scaling can allocate enough capacity over time, but because it operates at the timescale of minutes, it is too slow to resolve microsecond-scale imbalances. Furthermore, over-provisioning resources can be cost-inefficient if used to handle transient spikes in demand, such as those that occur during temporary failures [3].

Fairness. When the server has a sufficient number of credits, it tries to approximate max-min fairness when distributing credits to clients. However, when the number of available credits is less than the number of clients, Breakwater does not provide fairness to clients. Instead, it favors clients for which it is currently processing requests. This allows the server to piggyback credits to the responses and avoid sending explicit

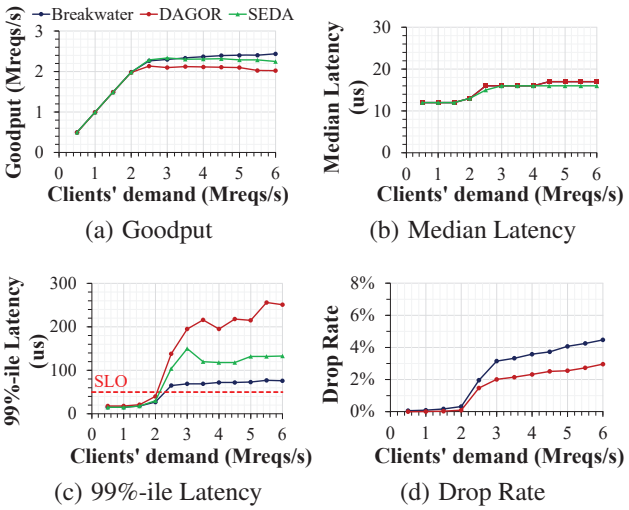


Figure 12: Memcached performance for USR workload with 10,000 clients (SLO = 50 µs)

credit messages. This preference toward a subset of clients is common in production services [51]. If a service operator wants to provide fairness among clients, the clients receiving the most credits could be timed-out over a longer timescale, so clients starved of credits can get a chance to send instead.

Overload control for multi-layer services. In this paper, we only consider a single-layer, single-server overload control scenario. Breakwater's receiver-driven, credit-based approach can be applied to multiple layers of microservices, preventing overload at each individual layer. However, when an overload occurs in an intermediate layer of a multi-layer service, the work performed in earlier layers is wasted. We leave propagating overload signals and coordinating overload control across several layers of microservices for future work.

7 Related Work

Receiver-driven transport protocols. Homa [34], NDP [24], and ExpressPass [14] schedule network packets with a receiver-driven mechanism to achieve high throughput and low latency. While Homa and Breakwater share some similarities including a credit-based, receiver-driven scheme and credit overcommitment, they are different in three significant aspects. First, Homa handles network congestion, whereas Breakwater handles server overload, which means that Breakwater must handle the additional challenges posed by overload control discussed in §2.3. Second, Homa relies on full knowledge of clients' demand, whereas Breakwater does not. Instead, the Breakwater server speculates clients' demand based on the latest demand information, the number of clients, and the number of available credits to minimize the message overhead. Third, both the motivation and the mechanism of overcommitment are different. Homa overcommits a fixed number of credits to handle an all-to-all workload, where a sender may get credits from multiple receivers and therefore not be able to send to all of them

simultaneously. In Breakwater, however, the server does not know which clients have demand. Thus, it dynamically increases the amount of overcommitted credits until it receives sufficient requests to keep itself busy with demand speculation.

Transport protocol for µs-scale RPCs. R2P2 [28] is a request/response-aware transport protocol designed for µs-scale RPCs. It implements JBSQ inside a programmable switch to better load balance requests among multiple servers. R2P2 limits the number of requests in a server's queue by explicitly pulling the requests from the switch. Through this mechanism, R2P2 provides bounded request queueing and low tail latency when the clients' demand is less than the servers' capacity. However, R2P2 does not provide any server overload control mechanism. If the clients' demand exceeds the servers' capacity, the request queue will build up at the switch, causing requests to violate their SLO. SVEN [27] builds upon R2P2 by adding a server overload control mechanism. Specifically, it drops requests at the switch if sampled tail latency exceeds an SLO-derived threshold. SVEN avoids the cost of request drops at the server by dropping requests early at the switch. However, unlike Breakwater, message overhead increases as clients' demand increases.

Circuit breaker in proxy. Envoy [6], HAProxy [7], NGINX [44], and GateKeeper [21] provide circuit breaker mechanisms to prevent back-end server overload. These proxies sit in front of a back-end server and stop forwarding requests to the server when one of the load metrics (e.g., the number of connections, the number of outstanding requests, the response time, estimated load) exceeds a threshold. However, since those thresholds must be set manually, it's challenging to find the right threshold value that maximizes resource utilization while keeping latency low.

Server overload control. Session-based admission control [12, 13] prevents web server overloads by limiting the creation of new sessions based on the number of successfully completed sessions or QoS metrics. However, they are not compatible with request-response models as they cannot prevent server overloads caused by a single session from a proxy that forwards requests from multiple clients. CoDel [38] controls the queuing delay of a server to prevent server overloads. Still, if the incoming packet rate is high and CPU is used more for packet processing, the server becomes less CPU efficient and degrades throughput. ORCA [29], SEDA [48], and Doorman [4] rate limit clients so that their sending rates do not exceed the server capacity. Doorman requires manually setting of the server capacity threshold. Both ORCA and SEDA may suffer from long queueing delays or underutilization if clients make mistakes on their sending rate with stale congestion information from the server. DAGOR [51] takes a hybrid approach using both AQM and client-side rate limiting using adaptive parameter based on queueing delay. However, as DAGOR server updates congestion status with responses, clients still can undershoot or overshoot the server capacity with stale information on server congestion when client demand is sporadic.

Flow control. TCP flow control prevents the sender from transmitting more bytes than the receiver can accommodate. The objective of TCP flow control is to avoid memory overrun at the server, not to prevent server overload or SLO violations. More recently, an SLO-aware TCP flow control mechanism [26] was proposed where the server adjusts receive window size in TCP header based on SLO and the queueing delay at the server. This approach limits the "bytes" of the incoming requests to prevent server overload, but it's challenging to decide the appropriate receive window size, especially when the request size is variable.

8 Conclusion

In this paper, we presented Breakwater, a server-driven, credit-based overload control system for microsecond-scale RPCs. Breakwater achieves high throughput and low latency regardless of the RPC service time, the load at the server, and the number of clients generating the load. Breakwater generates credits based on queueing delay at the server, maintaining high utilization by targeting non-zero queueing delay while avoiding queue buildup. To minimize the overhead of coordination between the clients and the server, we propose demand speculation and credit overcommitment to realize the credit-based design for overload control with minimal overhead. By estimating clients' demand and issuing more credits than their capacity, Breakwater eliminates the extra messaging cost which is often required with a credit-based approach. Additionally, Breakwater reduces its remaining messaging overhead significantly by piggybacking demands and credits to requests and responses, respectively. Our evaluation of Breakwater shows that it outperforms state-of-the-art overload control systems. In particular, Breakwater achieves 25× faster convergence with 6% higher converged goodput than DAGOR and 79× faster convergence with 3% higher converged goodput than SEDA when the clients' demand suddenly spikes to 1.4× capacity.

Acknowledgments

We thank our shepherd Rachit Agarwal and the anonymous reviewers for their valuable feedback, and Cloudlab [20] for providing us with infrastructure for development and evaluation. We also thank the anonymous artifact evaluators for verifying our artifacts. This work was supported by the Cisco Research Center Award, NSF grants (CNS-1563826, CNS-1751009, and CNS-1910676), a Facebook Research Award, a Microsoft Faculty Fellowship, and a VMWare Systems Research Award.

References

[1] High-performance, feature-rich netxtreme® e-series dual-port 100g pcie ethernet nic. https://www.broadcom.com/products/

ethernet-connectivity/network-adapters/100gb-nic-ocp/p2100g.

[2] Memcached. http://memcached.org/.

[3] More on today's gmail issue, 2009. https://gmail.googleblog.com/2009/09/more-on-todays-gmail-issue.html.

[4] Doorman: Global distributed client side rate limiting., 2016. https://github.com/youtube/doorman.

[5] AWS Auto Scaling, 2020. https://aws.amazon.com/autoscaling/.

[6] Envoy Proxy, 2020. https://www.envoyproxy.io/.

[7] HAProxy, 2020. http://www.haproxy.org/.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (DCTCP). In *SIGCOMM*, 2010.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[10] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.

[12] H. Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *INFOCOM*, 2002.

[13] L. Cherkasova and P. Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[14] I. Cho, K. Jang, and D. Han. Credit-scheduled delay-bounded congestion control for datacenters. In *SIGCOMM*, 2017.

[15] J. Cloud. Decomposing twitter: Adventures in service-oriented architecture. In *QCon New York*, 2013.

[16] A. Cockroft. Microservices workshop: Why, what, and how to get there. http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.

[17] A. Daglis, M. Sutherland, and B. Falsafi. RPCValet: Ni-driven tail-aware balancing of μs-scale rpcs. In *ASPLOS*, 2019.

[18] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[19] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *NSDI*, 2014.

[20] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of cloudlab. In *ATC*, 2019.

[21] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *International conference on World Wide Web*, 2004.

[22] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*, 2019.

[23] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *ICAC*, 2014.

[24] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, 2017.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, 2014.

[26] M. Kogias and E. Bugnion. Flow control for latency-critical rpcs. In *KBNets*, 2018.

[27] M. Kogias and E. Bugnion. Tail-tolerance as a systems principle not a metric. In *APNet*, 2020.

[28] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *ATC*, 2019.

[29] B. C. Kuszmaul, M. Frigo, J. M. Paluska, and A. S. Sandler. Everyone loves file: File storage service (FSS) in oracle cloud infrastructure. In *ATC*, 2019.

[30] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, 2015.

[31] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *GridCom*, 2010.

[32] B. Maurer. Fail at scale. *Queue*, 2015.

[33] J. C. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 1997.

[34] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, 2018.

[35] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. Acceltcp: Accelerating network applications with stateful TCP offloading. In *NSDI*, 2020.

[36] D. Namiot and M. Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2014.

[37] NGINX Documentation: Limiting Access to Proxied HTTP Resources, 2020. https://docs.nginx.com/nginx/admin-guide/security-controls/controlling-access-proxied-http.

[38] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 2012.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI*, 2013.

[40] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[41] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The RAMCloud storage system. *ACM Transactions on Computer Systems*, 2015.

[42] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *SOSP*, 2017.

[43] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: core-aware thread management. In *OSDI*, 2018.

[44] W. Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008.

[45] A. Sriraman and T. F. Wenisch. μtune: Auto-tuned threading for OLDI microservices. In *OSDI*, 2018.

[46] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu. Distributed resource management across process boundaries. In *SoCC*, 2017.

[47] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[48] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *SIGOPS European Workshop*, 2002.

[49] M. Welsh and D. E. Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[50] T. Zhang, J. Wang, J. Huang, J. Chen, Y. Pan, and G. Min. Tuning the aggressive tcp behavior for highly concurrent http connections in intra-datacenter. *Transactions on Networking*, 2017.

[51] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang. Overload control for scaling wechat microservices. In *SoCC*, 2018.

[52] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM*, 2015.



AIFM: High-Performance, Application-Integrated Far Memory

Zhenyuan Ruan Malte Schwarzkopf † Marcos K. Aguilera ‡ Adam Belay
MIT CSAIL *†Brown University* *‡VMware Research*

Abstract. Memory is the most contended and least elastic resource in datacenter servers today. Applications can use only local memory—which may be scarce—even though memory might be readily available on another server. This leads to unnecessary killings of workloads under memory pressure and reduces effective server utilization.

We present *application-integrated far memory* (AIFM), which makes remote, "far" memory available to applications through a simple API and with high performance. AIFM achieves the same common-case access latency for far memory as for local RAM; it avoids read and write amplification that paging-based approaches suffer; it allows data structure engineers to build *remoteable*, hybrid near/far memory data structures; and it makes far memory transparent and easy to use for application developers.

Our key insight is that exposing application-level semantics to a high-performance runtime makes efficient remoteable memory possible. Developers use AIFM's APIs to make allocations remoteable, and AIFM's runtime handles swapping objects in and out, prefetching, and memory evacuation.

We evaluate AIFM with a prototypical web application frontend, a NYC taxi data analytics workload, a memcached-like key-value cache, and Snappy compression. Adding AIFM remoteable memory to these applications increases their available memory without performance penalty. AIFM outperforms Fastswap, a state-of-the-art kernel-integrated, paging-based far memory system [6] by up to 61×.

1 Introduction

Memory (RAM) is the most constrained resource in today's datacenters. For example, the average memory utilization on servers at Google [73] and Alibaba [46] is 60%, with substantial variance across servers, compared to an average CPU utilization of around 40%. But memory is also the most inelastic resource: once a server runs out of available memory, some running applications must be killed. In a month, 790k jobs at Google had at least one instance killed, in many cases due to memory pressure [73]. A killed instance's work and accumulated state are lost, wasting both time and energy. This waste happens even though memory may be available on other servers in the cluster, or even locally: around 30% of server memory are "cold" and have not been accessed for minutes [41], suggesting they could be reclaimed.

Operating systems today support memory elasticity primarily through *swap* mechanisms, which free up RAM by pushing unused physical memory pages to a slower tier of memory, such as disks or remote memory. But OS swap mechanisms operate at a fixed and coarse granularity and incur substantial overheads. To swap in a page, the OS must handle a page fault, which requires entering the kernel and waiting until the data arrives. Figure 1 shows the throughput a recent page-based far memory system (*viz.*, Fastswap [6]) achieves when accessing remote objects using up to four CPU cores. Kernel swapping happens at the granularity of 4KB pages, so page-based far memory suffers read/write amplification when accessing small objects, as at least 4KB must always be transferred. Moreover, the Linux kernel spins while waiting for data from swap to avoid the overheads of context switch and interrupt handling. That means the wait time (about 15–20k cycles with Fastswap's RDMA backend) is wasted.

Throughput [accesses/sec]	*64B object*	*4KB object*
Paging-based (Fastswap [6])	582K	582K
AIFM	3,975K	1,059K

Figure 1: AIFM achieves 6.8× higher throughput for 64B objects and 1.81× higher throughput for 4KB objects, compared to Fastswap [6], a page-granular, kernel-integrated far memory approach. AIFM performs well since it (*i*) avoids IO amplification and (*ii*) context switches while waiting for data.

We describe a fundamentally different approach: *application-integrated far memory* (AIFM), which ties swapping to individual application-level memory objects, rather than the virtual memory (VM) abstraction of pages. Developers write *remoteable* data structures whose backing memory can be local and "far"—*i.e.*, on a remote server—without affecting common-case latency or application throughput. When AIFM detects memory pressure, its runtime swaps out objects and turns all pointers to the objects into remote pointers. When the application dereferences a remote pointer, a lightweight green threads runtime restores the object to local memory. The runtime's low context switch cost permits other green threads to make productive use of the wait cycles, which hides remote access latency and maintains high throughput. Due to these fast context switches, AIFM achieves 81% higher throughput than page-based approaches when accessing 4KB objects, and because AIFM avoids amplification, it achieves 6.8× higher throughput for small objects (Figure 1).

AIFM's programming interface is based on four key ideas: a fast, low-overhead *remoteable pointer* abstraction, a pauseless memory evacuator, runtime APIs that allow data struc-

tures to convey semantic information to the runtime, and a *remote device* interface that helps offload light computations to remote memory. These AIFM APIs allow data structure engineers to build hybrid local/remote data structures with ease, and provide a developer experience similar to C++ standard library data structures. The pauseless memory evacuator ensures that application threads never experience latency spikes due to swapping. Because data structures convey their semantics to the runtime, AIFM supports custom prefetching and caching policies—*e.g.*, prefetching remote data in a remoteable list and streaming of remote data that avoids polluting the local memory cache. Finally, AIFM's offloading reduces data movement and alleviates the network bottleneck that most far-memory systems experience.

The combination of these ideas allows AIFM to achieve object access latencies bounded only by hardware speed: if an object is local, its access latency is comparable to an ordinary pointer dereference; when it is remote, AIFM's access latency is close to the hardware device latency.

We evaluate AIFM with a real-world data analytics workload built on DataFrames [16], a synthetic web application frontend that uses several remoteable data structures, as well as a memcached-style workload, Snappy compression, and microbenchmarks. Our experiments show that AIFM maintains high application request throughput and outperforms a state-of-them-art, page-based remote memory system, Fastswap, by up to 61×. In summary, our contributions are:

1. Application-integrated far memory (AIFM), a new design to extend a server's effective memory size using "far" memory on other servers or storage devices.
2. A realization of AIFM with convenient APIs for development of applications and remoteable data structures.
3. A high-performance runtime design using green threads and a pauseless memory evacuator that imposes minimal overhead on local object accesses and avoids wasting cycles while waiting for remote object data.
4. Evaluation of our AIFM prototype on several workloads, and microbenchmarks that justify our design choices.

Our prototype is limited to unshared far memory objects on a single memory server. Future work may add multi-server support, devise strategies for dynamic sizing of remote memory, or investigate sharing.

2 Background and Related Work

OS swapping and far memory. Operating systems today primarily achieve memory elasticity by *swapping* physical memory pages out into secondary storage. Classically, secondary storage consisted of disks, which are larger and cheaper but slower than DRAM. The use of disk-based swap has been rare in datacenters, since it incurs a large performance penalty. More recent efforts consider swapping to a faster tier of memory or *far memory*, such as the remote memory of a host [3, 6, 21, 27, 28, 31, 40, 45, 48, 67] or a compression cache [24, 41, 81, 82]. Since swapping is integrated with the kernel virtual memory subsystem, it is transparent to user-space applications. But this transparency also forces swapping granularity to the smallest virtual memory primitive, a 4KB page. Combined with memory objects smaller than 4KB, this leads to *I/O amplification*: when accessing an object, the kernel must swap in a full 4KB page independent of the object's actual memory size. Moreover, supplying application semantic information, such as the expected memory access pattern, the appropriate prefetch strategy, or memory hotness, is limited to coarse and inflexible interfaces like `madvise`.

AIFM uses far memory in a different way from swapping, by operating at *object granularity* rather than page-granularity—an idea that we borrow from prior work on distributed shared memory (see below), memory compression [75], and SSD storage [1]. These investigations all point to page-level I/O amplification as a key motivation.

AIFM provides transparent access to far memory using smart pointers and dereference scopes inspired by C++ weak pointers [69], and Folly RCU guards [26].

Disaggregated and distributed shared memory. Disaggregated memory [58] refers to a hardware architecture where a fast fabric connects hosts to a pool of memory [29, 33], which is possibly managed by a cluster-wide operating system [33, 66]. Disaggregated memory requires new hardware that has not yet made it to production. AIFM focuses on software solutions for today's hardware.

Distributed shared memory (DSM) provides an abstraction of shared memory implemented over message passing [7, 10, 44, 50, 64, 65]. Like far memory, DSM systems can be page-based or object-based. DSM differs from far memory both conceptually and practically. Conceptually, DSM provides a different abstraction, where data is *shared* across different hosts (the "S" in DSM). Practically, this abstraction leads to complexity and inefficiency, as DSM requires a cache coherence protocol that impairs performance. For instance, accessing data must determine if a remote cache holds a copy of the data. By contrast, data in far memory is private to a host—a stricter abstraction that makes it possible to realize far memory more efficiently. Finally, DSM systems were designed decades ago, and architectural details and constants of modern hardware differ from their environments.

Technologies to access remote data. TCP/IP is the dominant protocol for accessing data remotely, and AIFM currently uses TCP/IP. Faster alternatives to TCP/IP exist, and could be used to improve AIFM further, but these technologies are orthogonal or complementary to AIFM's key ideas.

RDMA is an old technology that has recently been commoditized over Ethernet [32], generating new interest. Much work is devoted to using RDMA efficiently in general [39, 51, 76] or for specific applications, such as key-value stores (*e.g.*, [38, 49]) or database systems [11]. Smart NICs use CPUs or FPGAs [47, 52, 70] to provide programmable remote functionality [18, 43, 68]. AIFM requires no specialized hardware.

Abstractions for remote data. Remote Procedure Calls (RPCs) [12] are widely used to access remote data, including over RDMA [19, 71] or TCP/IP [37]. Memory-mapped files can offer remote memory behind a familiar abstraction [2, 67], while data structure libraries for remote data [4, 15], offer maps, sets, multisets, lists, and other familiar constructs to developers. This is similar in spirit to data structure libraries for persistent memory [59, 62]. AIFM offers a lower-level service that helps programmers develop such data structures.

I/O amplification. As mentioned, page-based access leads to I/O amplification, a problem studied extensively in the context of storage systems [1, 61] and far-memory systems [17], where hardware-based solutions can reduce amplification by tracking accesses at the granularity of cache lines.

Garbage collection and memory evacuation. Moving objects to remote memory in AIFM ("evacuation") is closely related to mark-compact garbage collection (GC) in managed languages. The main difference is that AIFM aims to increase memory capacity by moving cold, but live objects to remote memory, while GCs focus on releasing dead, unreferenced objects' memory. AIFM uses referencing counting to free dead objects, avoiding the need for a tracing stage. Instead of inventing a new evacuation algorithm, AIFM borrows ideas from the GC literature and adapts them to far-memory systems. Like GCs, AIFM leverages a read/write barrier to maintain object hotness [5, 14, 34], but AIFM uses a one-byte hotness counter instead of a one-bit flag, allowing more fine-grained replacement policies. Like AIFM, some copying collectors optimize data locality by separating hot and cold data during GC, but target different memory hierarchies; *e.g.*, the cache-DRAM hierarchy [34], the DRAM-NVM hierarchy [5, 79, 80], and the DRAM-disk hierarchy [14]. Finally, memory evacuation interferes with user tasks and impacts their performance. To reduce the interference, AIFM adopts an approach similar to the pauseless GC algorithms in managed languages [20], as opposed to the stop-the-world GC algorithms [36].

3 Motivation

Kernel paging mechanisms impose substantial overheads over the fundamental cost of accessing far memory.

Consider Figure 2, which breaks down the costs of Linux (v5.0.0) retrieving a swapped-out page from an SSD. The device's hardware latency is about 6µs, but Linux takes over 15µs (2.5×) due to overheads associated with locking (P1, P5), virtual memory management (P2, P3, P5), accounting (P4), and read IO amplification (P3). Moreover, due to the high cost of context switches, Linux spins while waiting for data (P3), wasting 11.7µs of possible compute time.

AIFM, by contrast, provides low-overhead abstractions and an efficient user-space runtime that avoid these costs, bringing its latency (6.8µs) close to the hardware limit of 6µs. We explain these concepts in the next two sections.

Phase	Linux Kernel Swapping	AIFM
P1	Page fault, trap to kernel	Deref far pointer, issue I/O
P2	Lock, get PTE, allocate page frame, allocate swap cache entry	Lightweight context-switch
P3	Issue read I/O, spin, insert PFN in global LRU list	Run another green thread
P4	cgroup accounting, reclaim memory if past limit	I/O completion, context-switch back
P5	Set page mapping, unlock	—

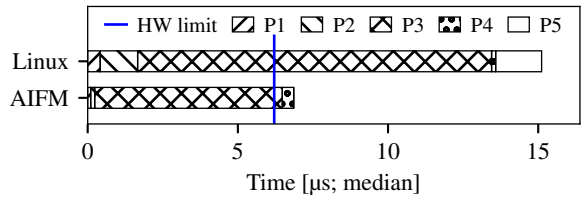


Figure 2: Linux kernel-based swapping has high overheads over hardware I/O limits (blue line, 6µs). Both Linux and AIFM use an SSD device backend in this experiment.

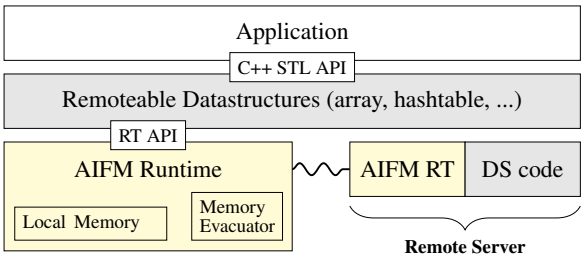


Figure 3: Applications use remoteable data structures (gray), and data structure developers rely on the AIFM runtime (yellow) to handle local memory management and interact with remote memory. Data structures can have active remote components (*i.e.*, the "DS code" box) to offload light computation.

4 AIFM Design

The goal of Application-Integrated Far Memory (AIFM) is to provide an easy-to-use, efficient interface for far memory without the overheads of page-granular far memory.

4.1 Overview

AIFM targets two constituencies: application developers and data structure developers. AIFM provides application developers with data structures with familiar APIs, allowing developers to treat these *remoteable* data structures mostly as black boxes; and AIFM provides simple, but powerful APIs to data structure engineers, allowing them to implement a variety of efficient remoteable memory data structures. Figure 3 shows a high-level overview of AIFM's design: applications interact with data structures (gray) implemented using primitives and APIs provided by the AIFM runtime (yellow).

For an **application developer**, programming applications that use far memory should feel almost the same as programming with purely local data structures. In particular, the developer should not need to be aware of whether an object is currently local or remote (*i.e.*, far memory is *transparent*), and remoteable memory data structures should offer the same

performance as local ones in the common case. For example, idiomatic C++ code for reading several hash table entries and an array element computed from them might look as follows:

```
std::unordered_map<key_t, int> hashtable;
std::array<data_t> arr;

void print_data(std::vector<key_t>& request_keys) {
  int sum = 0;
  for (auto key : request_keys) {
    sum += hashtable.at(key);
  }
  std::cout << arr.at(sum) << std::endl;
}
```

The same code written using AIFM looks like this:

```
RemHashtable<key_t, int> hashtable;
RemArray<data_t> arr;

void print_data(std::vector<key_t>& request_keys) {
  int sum = 0;
  for (auto key : request_keys) {
    DerefScope s1; // Explained in Section 4.2.2.
    sum += hashtable.at(key, s1);
  }
  DerefScope s2;
  std::cout << arr.at(sum, s2) << std::endl;
}
```

The remoteable memory data structures themselves (`RemHashtable` and `RemArray` above) are written by **data structure engineers**, who use AIFM's runtime APIs to include remoteable memory objects in their data structures. When memory becomes tight, AIFM's runtime moves some of these memory objects to remote memory; when the data structure needs to access remote objects, the AIFM runtime fetches them. Data structure engineers have substantial design freedom: they can rely entirely on AIFM to fetch remote objects, or they can deploy custom logic on the remote side.

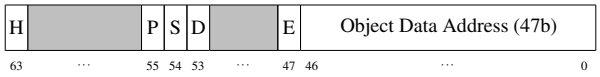
Remote servers store the actual remote data in their memory, and run a counterpart AIFM runtime, which may call into custom data structure logic. This is helpful, *e.g.*, if the remoteable memory data structure needs to chase pointers, which would otherwise require multiple round-trips.

4.2 Remoteable Memory Abstractions

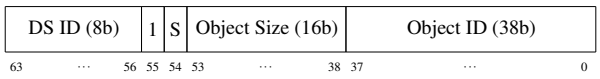
AIFM is designed around four core abstractions: *remoteable pointers*, *dereference scopes*, *evacuation handlers*, and *remote devices*. We designed the abstractions such that they impose minimal overheads (as low as three micro-ops) on "hot path" access to local objects, and try to ensure that the "cold path" remote access incurs little latency above hardware limits.

4.2.1 Remoteable Pointers

A remoteable pointer represents a memory object (*i.e.*, an allocation) that is currently either local, or remote (in "far" memory). AIFM supports unique and shared remoteable pointers, whose interface makes them suitable for use in any place where a data structure would use an ordinary, local pointer.



(a) Local object. H: hot, P: present, S: shared, D: dirty, E: evacuating.



(b) Remote (swapped-out) object. DS ID means data structure ID.

Figure 4: Remoteable unique pointer representations for local and remote objects. AIFM inverts the H/P/D bit meaning (0 = hot/present/dirty) for a more efficient hot path execution.

Memory representation. Unique remoteable pointers, which correspond to C++'s `std::unique_ptr`, have the same size as ordinary 64-bit pointers, while shared pointers are 128-bits wide (like `std::shared_ptr`). Figure 4 shows the memory layout of a remoteable unique pointer. Depending on whether a remoteable pointer is local or remote, we adopt a different format. If the memory is local (Figure 4a), the pointer contains a virtual memory address in its lower 47 bits (enough to represent user-space addresses), and control bits in the upper 17 bits, including standard dirty (D) and present (P) bits (cf. page tables). It also contains bits to track whether the pointer is hot (H) and whether it is being concurrently evacuated (E). For unique pointers, the shared (S) bit is set to 0. We byte-align the D, E, and H bits, allowing each of them to be accessed by mutators and runtime evacuators concurrently and atomically, as a byte is the smallest read/write unit.

If the memory is remote (Figure 4b), it contains metadata to assist in retrieving the object from remote memory, such as the data structure ID, the object size, and the object ID. Each data structure instance has a unique data structure ID managed by the runtime. The object ID refers to a data structure-specific object identifier (such as a key in a hash table), which is used by the remote memory server to identify the object.

AIFM's remoteable shared pointer, which allows pointer aliasing and corresponds to C++'s `std::shared_ptr` differs from the unique pointer in two ways. First, its S bit is set to 1; and second, the pointer has an additional 8 bytes for chaining the shared pointers to the same object. When AIFM's runtime evacuates the referred object or moves it locally (§5.3), it traverses the chain to update all shared pointers.

API. Listing 1 shows the API of the remoteable unique pointer (the shared pointer's API is largely identical). `RemUniquePtr` has two constructors: one for already-local objects and one for currently remote objects. The second constructor allows data structures to form remoteable pointers to objects that are currently remote. This helps data structure engineers reference remote objects from their data structures without having to fetch those objects.

To turn a remote pointer into a local one, the programmer dereferences it via the `deref` and `deref_mut` API methods.

```
class RemUniquePtr<T> {
  uint64_t metadata; // 64 bits, see Figure 4.
  // Construct local object
  RemUniquePtr(DSID, T* obj_addr);
  // Construct remote object
  RemUniquePtr(DSID, ObjID);
  const T* deref(DerefScope& scope); // Immutable.
  T* deref_mut(DerefScope& scope); // Mutable.
}
```

Listing 1: AIFM remoteable unique pointer API.

Dereferencing. When the dereferencing methods are called, the runtime inspects the present bit of the remoteable pointer. If the object is local, it sets the hot bit and returns the address stored in the pointer. Otherwise, the runtime fetches it from the remote server, sets the hot bit and dirty bit (in `deref_mut`), and returns a local pointer to the data.

AIFM's hot path for local access is carefully optimized and takes five x86-64 machine instructions: one `mov` to load the pointer, one `andl` to check present and evacuating bits, a conditional branch to the cold path if neither is set, a shift (`shrq`) to extract the object address, and a `mov` to return it. Modern x86-64 processors macro-fuse the second and third instructions (test and branch), so the hot path requires four micro-ops, a three-micro-op overhead over an ordinary pointer dereference. The cold path is slower, as it calls into the AIFM runtime to potentially swap in a remote object.

One challenge to making this API work is managing the local lifetime of the dereferenced data: while the application holds a pointer returned from dereferencing a `RemUniquePtr`, the runtime must never swap out the object. This is hard to achieve in unmanaged languages like C/C++, since after getting the raw address, application code could store it virtually anywhere (*e.g.*, on the heap, stack, or even in registers). The runtime lacks sufficient information to detect whether any such pointer continues to exist, and thus whether the data is still being used. The Boehm garbage collector [13] tackles a similar reference lifetime problem by scanning the whole address space to find any possible references. Such scans would impose an unacceptable performance overhead for AIFM. Our solution is to instead leverage application semantics to tie the lifetime of the local, dereferenced data to the lifetime of the AIFM's dereference scopes.

4.2.2 Dereference Scopes

Listing 2 demonstrates the usage of `DerefScope`. Before accessing the remoteable object, the developer must construct a `DerefScope`. AIFM container's API provides a compile-time check by taking a `DerefScope&` argument. (This is also why the remoteable pointer has its own dereferencing methods, rather than overloading `operator*`.)

Under the hood, `DerefScope`'s constructor creates an evacuation fence, which blocks upcoming evacuations until it is destructed. The lifetime of all local dereferenced data is therefore tied to the scope lifetime. Accessing dereferenced data

```
RemVector<value_t> vec;
// ...
for (uint64_t i = 0; i < vec.size(); i++) {
  {
    DerefScope scope;
    auto& value = vec->at(i, scope);
    // process value
  }
  // scope destroyed, can evacuate value's object
}
```

Listing 2: AIFM dereference scope example.

outside the dereference scope is undefined behavior. In the future, AIFM might leverage static analysis to catch lifetime violations, as in the Rust compiler [78].

Our scope API is familiar to C/C++ programmers; it shares similarity with C++11's `std::weak_ptr` and, *e.g.*, the `rcu_reader` guard in Facebook's RCU API [26]. Note that the lifetime of the `DerefScope` is separate from the lifetime of the remoteable pointer: a remoteable pointer may still be alive even when its data has been swapped to the remote. This is unlike, *e.g.*, `std::unique_ptr`, where the pointer's destructor terminates the lifetime of the object data.

Dereference scopes require developers to modify the application code. An alternative API might avoid the need for a dereference scope at the cost of copying the object into local memory on dereference. AIFM's core APIs aim to achieve maximum performance, so we avoid copying by default. The overhead of a copying API is highly application-dependent; our experiments suggest that 3–8% overhead are typical for applications with high compute/memory access ratios.

4.2.3 Evacuation Handlers

When an object is not protected by a `DerefScope`, AIFM's runtime may evacuate it to far memory. Evacuation changes the pointer to this object from local to remote status, and future dereferences will cause AIFM to swap the object back in. But some use cases may wish to implement custom behavior on evacuation. For example, when AIFM evacuates an object contained in a hash table, the hash table may register an evacuation handler to remove the key and object pointer to save local space. (In this case, future lookup misses for the key will reconstitute the key and pointer, and add them to the hash table.) AIFM offers *evacuation handlers* for this purpose, enabling developers to incorporate the data structure semantics into the runtime evacuator.

Evacuation handlers are also critical for handling embedded remoteable pointers inside objects. For example, data structure engineers can use evacuation handlers to support embedded remoteable unique pointers in objects that are themselves remoteable. When an object is remoted, any embedded remoteable pointers must either be moved to the local heap, or the object it references must be moved to remote memory, and the remoteable pointer must be updated with an identifier to later retrieve the remote object from a remote device (§4.2.4). As a result, the evacuator never has to retrieve remote memory

```
// The AIFM runtime will invoke the handler on evacuating
// the object to the remote server (phase 4 in Section 5.3).
using EvacHandler = std::function<
    void(Object&, const Runtime::CopyToRemoteFn&)>;
// Registers an evacuation handler for a data structure ID.
void Runtime::RegisterEvacHandler(DSID id, EvacHandler h);
```

Listing 3: AIFM evacuation handler API.

to update a remoteable pointer.

AIFM provides an evacuation handler API (Listing 3). The evacuation handler gets invoked on evacuating the object to the remote server (phase 4 in §5.3), right before the runtime frees the object's local memory. The runtime passes two arguments to `EvacHandler`—the object to be evacuated and the function that triggers the runtime to copy the object to the remote side. The first argument allows the handler to mutate the object data before copying (*e.g.*, modify the state of its embedded pointers) and further cleanup the local data structure after copying (*e.g.*, remove its pointer from the hash table index). The second argument offers the flexibility in the timing of copying the object to remote.

Data structure developers register their evacuation handlers by invoking `RegisterEvacHandler`. An evacuation handler is tied to a unique data structure ID, which each data structure allocates in its constructor, and which data structure engineers must use consistently. This way, different data structures or instances of the same data structure coexist in the same application, while the runtime invokes the appropriate handler.

4.2.4 Remote Devices

AIFM's `RemDevice` provides functionality at the remote memory server (Listing 4). The remote device, by default, uses a key-value store abstraction: when the client dereferences a remote pointer, the runtime sends the data structure ID and object ID to the remote server, which looks up the object by data structure ID and object ID, and sends the object data back. When evacuating an object, the runtime sends IDs and object data to the remote server, which inserts the object.

AIFM also gives datastructure engineers the flexibility to override this default behavior to integrate custom active components at the remote server. This is accomplished by registering their implementation on their own data structure type to the remote device (`register_active_component`). A custom active component is especially beneficial when the application's compute intensity is low, as this setting often makes it more efficient to perform operations on remote memory than paying the cost of bringing the objects into local memory. After registering the active component at the remote, data structure engineers invoke `RemDevice`'s client-side bindings to interact with the remote components. They use `construct` and `destruct` to instantiate and destroy remote components. If an object is not present when dereferencing a remote pointer, the runtime invokes the `read_obj` to swap in the missing object. On evacuation, the evacuator invokes `write_obj` to swap out cold objects and `delete_obj` to release dead objects. In

```
class RemDevice {
  void register_active_component(DSType, ActiveComponent&);
  DSID construct(DSType, ByteArray params);
  void destruct(DSID);
  void read_obj(DSID, ObjID, ByteArray& obj_data);
  void write_obj(DSID, ObjID, ByteArray obj_data);
  bool delete_obj(DSID, ObjID);
  void compute(DSID, OpCode, ByteArray in, ByteArray& out);
};
```

Listing 4: AIFM remoteable device API.

addition, the `compute` method invokes a custom function, executing a lightweight computation on the remote server. This is useful, for example, for efficiently aggregating a sum across objects in a data structure without wasting network bandwidth to bring all objects into local memory first.

We implemented remote active components to improve the performance of hashtables (§8.2.1) and DataFrames (§8.1.2).

4.2.5 Semantic Hints

AIFM's APIs allow injecting information about application- and object-specific semantics into the runtime.

Hotness tracking. To dereference a remoteable pointer, the user invokes our library, which sets the hot bit of the pointer. Under memory pressure, the memory evacuator uses this hotness information to ensure that frequently accessed objects are local. On evacuation, the evacuators clear the hot bit. AIFM initialization allows developers to customize the number of hot bits to use in the pointer (up to eight) and the replacement policy by data structure ID. With several hot bits, AIFM supports, *e.g.*, a CLOCK replacement policy [72].

Prefetching. AIFM includes a library that data structures can use to maintain a per-thread window of the history of dereferenced locations and predict future accesses using a finite-state machine (FSM). It updates the window and the FSM on each dereference. The FSM detects patterns of sequential access and strided access. When a pattern is detected, it starts prefetcher threads that swap in objects from the remote server. With enough prefetching, application threads always access local memory when dereferencing remoteable pointers. The library estimates the prefetch window size conservatively using the network bandwidth-delay product. Data structure engineers can also add custom prefetching policies.

Nontemporal Access[1]**.** For remoteable pointers to objects without temporal locality, it makes sense to limit the local memory used to store their object data. This avoids polluting local memory, which multiple data structures may share, with data that a data structure engineer knows is unlikely to be accessed again. To achieve this, AIFM's pointer API supports *non-temporal* dereferences (Listing 5). This immediately marks the object pointed to by `rmt_ptr` as reclaimable, though the actual evacuation happens only after the

[1] We use "nontemporal" in the sense of x86's nontemporal load/store instructions [35], which conceptually bypass the CPU cache to avoid pollution.

```
DerefScope scope;
// non-temporal dereference ⇓ allows immediate reclaim
T* p1 = rmt_ptr1.deref_mut<true>(scope);
// temporal deref; deref_mut(scope) works too
T* p2 = rmt_ptr2.deref_mut<false>(scope);
```

Listing 5: Non-temporal and temporal dereferences.

`DerefScope` ends. Without a hint, a dereference is temporal by default; §8.1.1 evaluates the benefit of the hint.

5 AIFM Runtime

AIFM's runtime is built on "green" threads (light-weight, user-level threading), a kernel-bypass TCP/IP networking stack, and a pauseless memory evacuator. Applications link the runtime into their user-space process. This allows us to co-design the runtime with AIFM's abstractions and provides high-performance far memory without relying on any OS kernel abstractions.

Two high-level objectives guide our runtime design: (*i*) the runtime should productively use the cycles spent waiting during the inevitable latency when fetching objects from remote memory; and (*ii*) application threads should never have to wait for the memory evacuator.

5.1 Hiding Remote Access Latency

We want to hide the latency of fetching data from far memory by doing useful work during the fetch.

Existing OS kernel threads pay high context-switching costs: *e.g.*, on Linux, rescheduling a task takes around 500ns. These costs are a nontrivial fraction of remote memory latency, so Linux and Fastswap adopt a design where they busy-spin while waiting for a network response [6]. This avoids context-switch overheads, but also wastes several microseconds of processing time. This approach also places tremendous pressure on network providers to support even lower latency to reduce the amount of wasted cycles [9, 28]. AIFM takes a different approach: it relies on low-overhead green threads to do application work while waiting for remote data fetches.

Consistent with literature on garbage collection (GC), we refer to normal application threads as *mutator threads* in the following. Each mutator thread accesses far memory, blocking whenever it needs to fetch a remote object. When that happens, another mutator thread can run and make productive use of available CPU cycles. Moreover, AIFM's runtime spawns prefetcher threads to pull in objects that it predicts will be dereferenced in the future, allowing it to avoid blocking mutator threads when the predictions are correct.

Using green threads, AIFM tolerates network latency without sacrificing application-level throughput, wasting fewer cycles than systems that busy-poll for network completion.

5.2 Remoteable Memory Layout

For the local memory managed by AIFM, its runtime embraces the idea of log-structured memory [63], which splits

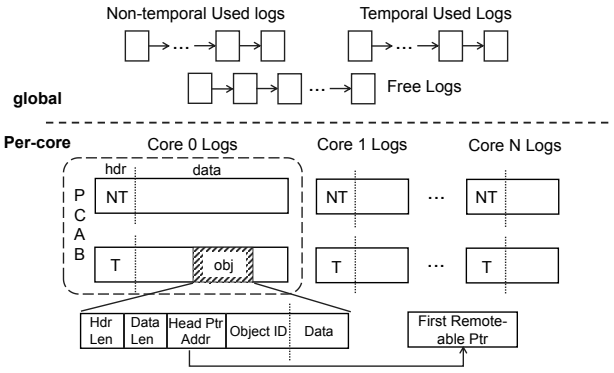


Figure 5: The layout of local remoteable memory in AIFM. There are three global lists: a free list, a temporal used list, and a non-temporal used list. Each list stores many logs, and each log stores many objects. There is a per-core allocation buffer (PCAB) that keeps two free logs to allocate new objects, one log for temporal objects, the other for non-temporal ones.

and manages the local remoteable memory in the granularity of logs (Figure 5). The log size is 2MB, which helps reduce TLB misses by allocating huge pages. The runtime maintains three global lists: a free list, a non-temporal used list, and a temporal used list. Each list stores many logs. For core scalability, each core owns two logs for new allocations: one log for temporal objects, the other for non-temporal ones. The logs are kept in a per-core allocation buffer (PCAB). To allocate an object, the runtime first tries to allocate from a log in the PCAB. If that log runs out of space, the runtime appends the log to the global non-temporal or temporal used list, and obtains a new log from the global free list. To free an object, the runtime marks the object as free. AIFM leverages a mark-compact evacuator to achieve a low memory fragmentation ratio, as shown with other copying log allocators [63].

A log has a 1B header indicating whether it stores non-temporal or temporal data. The remaining space stores objects. Each object has a `Hdr Len` bytes header and a `Data Len` bytes data. The 6-byte `Head Ptr Addr` stores the address of the remoteable pointer that points to the object. For a unique pointer, `Head Ptr Addr` stores the address of the only pointer; for a shared pointer, it stores the address of the first shared pointer in the chain. Dead objects have `Head Ptr Addr` set to `nullptr`. The variable-sized `Object ID` stores the object's unique identifier. The header is used on evacuation, when the runtime passes the object ID to write/delete endpoints on the remote device and the remoteable pointer address to the evacuation handler, and when the runtime swaps in an object and passes the object ID to the remote device.

5.3 Pauseless Memory Evacuator

Upon memory pressure, the runtime's memory evacuator moves cold objects to the remote server. Like with many garbage collectors in managed languages, a key feature of AIFM is to allow mutator threads to run concurrently while

the runtime evacuates local memory. The evacuator executes four phases in sequence, described in the following paragraphs. To ensure correctness under race conditions, the evacuator maintains an invariant: *it only starts to move object O after setting the mutator-side synchronization barrier on accessing O*. The evacuator sets the barrier by setting the pointer evacuation bit (phase 2). The RCU writer wait (phase 3) ensures all mutators have observed the set bits to enforce the timing order in the invariant.

1. Log Selection Phase. The goal of the evacuator is to maintain the local free memory ratio above the *min_free_ratio* (0.12 by default). The *master thread* of the evacuator picks $total_log_cnts \cdot (current_free_ratio - min_free_ratio)$ of logs to be evacuated. The evacuator picks logs in FIFO order from the global non-temporal used list, and then picks from the global temporal used list if necessary, to prioritize non-temporal objects. AIFM could also use more sophisticated schemes, *e.g.*, prioritizing logs by occupancy and age [23].

2. Concurrent Marking Phase. The master evacuation thread spawns *worker threads* and divides the previously-selected logs among them. Each worker thread iterates through the objects in its logs to find live objects. For each such object, the worker sets the evacuation bit of all remoteable pointers of the object by traversing the pointer chain starting from the head pointer address (i.e., the `Head Ptr Addr` field). This marks the object for evacuation.

3. Evacuator Waiting Phase. The runtime can evacuate objects only when they are not being dereferenced by mutator threads. Rather than following a naive approach of having mutators and the evacuator to acquire a per-object lock—which would impose high overhead on the hot path of mutators accessing local objects—AIFM uses an approach inspired by read-copy-update (RCU) synchronization. AIFM's runtime treats mutators as RCU readers and the evacuator master thread as an RCU writer, thereby moving the synchronization overhead to the evacuator. This choice makes sense because (*i*) the mutators do application work, so AIFM should steer overhead away from them; and (*ii*) evacuation is a rare event. The result is that the evacuator master thread waits for a quiescent period to ensure all mutator threads have witnessed the newly-set evacuation bits.

If a mutator thread subsequently dereferences a pointer to an object that the runtime is evacuating, the mutator sees that the evacuation bit is set. A naive approach would now block the mutator thread while the evacuation bit is set. Instead, AIFM opts for an approach that avoids such pauses: the mutator copies the object to another log in its PCAB, and then executes a compare-and-swap (CAS) on the head remoteable pointer (which serves as a synchronization point) to simultaneously clear the evacuation bit, set the present bit, and set the new data location. This CAS will race with the evacuator (see next phase below). If the CAS succeeds, the mutator copied an intact object, so it obtains a local reference. The mutator then updates all pointers in the pointer chain with the head pointer metadata and continues executing. If the CAS fails, the evacuator has already changed the remoteable pointer to remote status, so the mutator's copy of the object may be corrupt. Consequently, the mutator frees the copy it made and obtains a remote reference.

4. Concurrent Evacuation Phase. The master thread spawns more worker threads to evacuate objects and run their evacuation handlers. Again, the master divides the previously selected logs among the workers. Each worker iterates through each log and each object within the log. For each cold object, the worker copies the object to the remote and executes a CAS on the head remoteable pointer to simultaneously clear the presence bit and set the remote pointer metadata. If the CAS succeeds, the object has been evacuated, and the worker updates all pointers in the pointer chain with the head pointer metadata and invokes the evacuation handler. Otherwise, a mutator thread succeeded with a racing CAS and has copied the object to another location. Either way, the log entry is now unused and reclaimable. For each hot object, the worker compacts and copies it into a new log, updates the object address in the remoteable pointers, and resets the hot bits.

5.4 Co-design with the Thread Scheduler

Evacuation is an urgent task when the runtime is under memory pressure. With a naive thread scheduler, evacuation can be starved by mutator threads, leading to out-of-memory errors and application crashes. There are two challenges that we need to address. First, a large number of mutator threads may allocate memory faster than evacuation can free memory. Second, evacuation sometimes blocks on mutator threads in a dereference scope, and this creates a dilemma. On one hand, the scheduler needs to execute mutator threads so they can unblock evacuation. On the other hand, executing mutator threads may consume more memory.

To address these issues, we co-design the runtime's green thread scheduler with AIFM to prioritize the activities necessary for evacuation, both in mutator threads and evacuation threads. First, each thread keeps a status field that is set by the AIFM runtime and read by the scheduler, which allows the scheduler to know whether a thread is in a dereference scope. The scheduler runs a multi-queue algorithm and assigns the first priority to mutators in a dereference scope, second priority to evacuation threads, and third priority to other mutator threads. Second, to avoid priority inversion [42] when the system is short of memory, the allocation function in the AIFM runtime triggers a signal to all running threads to force them to yield their cores back to the scheduler for re-scheduling.

6 Remoteable Data Structure Examples

We implemented six remoteable AIFM data structures.

Array. The remoteable array consists of a native array of `RemUniquePtr`s. Each pointer points to an array element to enable fine-grained data placement decisions. Alternatively, users can configure the pointed object as multiple consecutive

array elements to reduce the memory overhead of pointer metadata. The object IDs of pointers are their remote-side object addresses. The prefetcher records accessed indices at all array access APIs; it starts prefetching when detecting a strided access pattern.

Vector. The remoteable vector is similar to the remoteable array except that it is dynamically sized, and uses a `std::vector` to store `RemUniquePtrs`. Additionally, the vector has an active remote component that supports offloading operations like copies and aggregations, which are used by the DataFrame application (§8.1.2).

List. The remoteable list is similar to the remoteable vector, except that it uses a local list that stores `RemUniquePtrs` to support efficient `insert` and `erase` operations. The list supports traversals in forward and reverse directions, which offers strong semantic hints to the prefetcher. When detecting a direction, the prefetcher walks through the local list in the same direction to prefetch remote list objects.

Stack and Queue. The remoteable stack and queue are simple wrappers around remoteable lists.

Hashtable. The remoteable hashtable consists of a table index (stored on the local heap) and key-value data (stored in AIFM's remoteable heap). In the index, each hash bucket stores a `RemUniquePtr` to a key-value object. The object IDs of pointers are their hashtable keys. The hashtable has an active remote component that maintains a separate hashtable in remote memory. In this architecture, the local hashtable is a cache (inclusive or exclusive) of its remote counterpart. When the referenced object is missing from the local cache, the active remote component assists the chain lookup at the remote hashtable to avoid multiple network round-trips. Data structure engineers might also realize different hashtable designs via AIFM's APIs.

7 Implementation

AIFM's implementation consists of the core runtime library (§5) and the data structure library (§6). The core runtime is built on top of Shenango [55] to leverage its fast user-level threading runtime and I/O stack. AIFM is written in C and C++, with 6,451 lines in the core runtime, 5,535 lines in the data structure library, and 750 lines of modifications to the Shenango runtime. The system runs on unmodified Linux.

We integrated two far memory backends into AIFM: a remote memory server based on a DPDK-based TCP stack, and an NVMe SSD using an SPDK-based storage stack. Unlike the remote memory backend, the SSD backend does not support active remote components (since the storage drive does not have a general compute unit), and it has an inherent I/O amplification because it is limited to a fixed block size. Our evaluation focuses on the remote memory backend.

The current implementation has some limitations. First, we do not support TCP offloading or RDMA, which would reduce CPU overhead of our runtime. Second, a local compute server connects to a single remote memory server, and the remote memory cannot be shared by different clients. Finally, the local and remote memory size cannot be changed at runtime. We plan to address them in the future.

8 Evaluation

Our evaluation of AIFM seeks to answer three questions:

1. What performance does AIFM achieve for end-to-end applications, including ones that combine multiple remoteable data structures? (§8.1)
2. How does AIFM's performance compare to a state-of-the-art far memory system, Fastswap [6]? (§8.1–§8.2)
3. What factors contribute to AIFM's performance? (§8.3)

Setup. We run experiments on two `xl170` nodes on CloudLab [25] with 10-core Intel Xeon E5-2640 v4 CPUs (2.40 GHz), 64GB RAM, and a 25 Gbits/s Mellanox ConnectX-4 Lx MT27710 NIC. We enabled hyper-threads, but disabled CPU C-states, dynamic CPU frequency scaling, transparent huge pages, and kernel mitigations for speculation attacks in line with prior work [55]. We use Ubuntu 18.04.3 (kernel v5.0.0) and DPDK 18.11.0, except for experiments with Fastswap, which use Linux kernel v4.11, the latest version Fastswap supports. All AIFM experiments use the default configuration settings and the default built-in prefetchers of remoteable data structures. We do not tune prefetching policy specifically for evaluated applications.

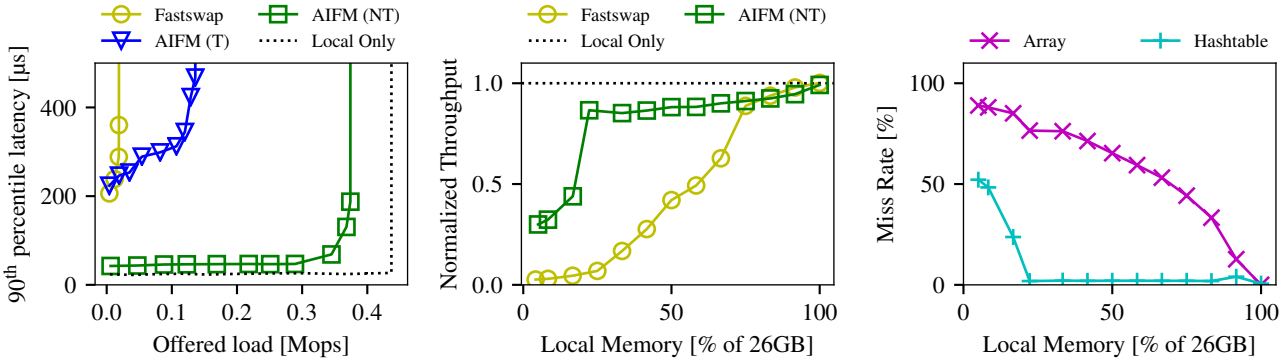
8.1 End-to-end Performance

We evaluate AIFM's end-to-end performance with two applications. First, we designed a synthetic application that mimics a typical web service frontend to understand AIFM's performance with multiple remoteable data structures and the impact of semantic hints. Second, we also ported an open-source C++ DataFrame library [16] with an interface similar to Pandas [56] to AIFM, and use it to understand the porting effort required and AIFM's performance for an existing application.

8.1.1 Synthetic Web Service Frontend

In response to client requests, the application fetches structured data (*e.g.*, a list of user IDs) from an in-memory key-value store, and then uses the retrieved values to compute an index into a large collection of 8KB objects (*e.g.*, profile pictures). Finally, the application fetches one 8KB object, encrypts it, and compresses it for the response to the client.

This application uses our remoteable hashtable (for the key-value pairs) and our remoteable array (for the 8KB objects). Each client request looks up 32 keys in the hashtable and fetches a single 8KB array element. We load the hashtable with 128M key-value pairs (10GB total data, of which 6GB are index data and 4GB are value data), and create an array of 2M objects of 8KB each (16GB total). The two data structures share 5GB of available local memory, *i.e.*, the local memory size is 19% of the total data set size. We generate closed-loop client requests from a Zipf distribution with parameter *s*: a uni-



(a) Latency as a function of offered load. **(b)** Throughput as a function of local memory. **(c)** AIFM miss rates in Figure 6b.

Figure 6: In a web frontend-like application with a hashtable and array, AIFM outperforms Fastswap by 20× (a) and achieves 90% of local memory performance with 5× less memory (b), as non-temporal array access avoids polluting local memory (c). "AIFM(NT)": non-temporal access; "AIFM(T)": temporal access; "Local Only": entire working set in local memory.

form distribution corresponds to $s = 0$, while values of s close to 1 indicate high skew. Each request accesses Zipf-distributed keys in the hashtable and uses their values to calculate an (also Zipf-distributed) array index to access; the request then encrypts the array data via AES-CBC using crypto++ [22] and compresses the result using Snappy [30]. We compare two AIFM settings—with and without non-temporal dereferences for array elements—against Fastswap [6] and an idealized baseline with all 26GB in local memory. A good result for AIFM would show improved performance over Fastswap, a benefit to non-temporal array accesses, and performance not much lower than keeping the entire data in local memory.

Figure 6a shows a throughput-latency plot for a Zipf parameter of $s = 0.8$ (*i.e.*, a skewed distribution). The x-axis shows the offered load in the system, and the y-axis plots the measured 90th percentile latency. Each setup eventually encounters a "hockey-stick" when it can no longer keep up with the offered load. Fastswap tolerates a load of up to 19k requests/second, but its overheads and the amplification for the hashtable lookups quickly dominate. AIFM with a temporal array dereference scales 7× further, but fails to keep up beyond 140k requests/second because the 8KB array accesses pollute its local memory. To make room for an 8KB array element, the runtime often evicts hundreds of hashtable entries, causing a high miss rate on hashtable lookups. AIFM with non-temporal access to the array, however, scales to 370k requests/second (20× Fastswap's maximum throughput). This is 16% lower throughput than the 440k requests/second achieved by an idealized setup with 26GB in local memory. In other words, AIFM achieves 84% of the performance of an entirely local setup with 5× less local memory.

Additional local memory helps bring AIFM performance closer to the in-memory ideal. Figure 6b shows the percentage of the all-local memory throughput achieved by the non-temporal version of AIFM when varying the local memory size (on the x-axis, as a fraction of 26GB). While Fastswap's throughput starts near zero and grows roughly in proportion to the local memory size, AIFM's throughput starts at 30% of the ideal and quickly reaches 85% of the in-memory throughput at 5.0GB local memory (20% of 26GB).

Figure 6c illustrates why this happens. At the left-hand side of the plot (5% local memory), AIFM sees high miss rates in both hashtable (52%) and array (89%). But as local memory grows, the hashtable miss rate quickly drops to near-zero, since AIFM's non-temporal dereferences for the array ensure that most of the local memory is dedicated to hash table entries. Correspondingly, the array miss rate drops more slowly and in proportion to the local memory available. By contrast, Fastswap (not shown here) has high miss rates in both data structures, as its page-granular approach manages local memory inefficiently.

8.1.2 DataFrame Application

The DataFrame abstraction, popularized in Pandas [56], provides a convenient set of APIs for data science and ML workloads. A DataFrame is a table-structured, in-memory datastructure exposing various slicing, filtering, and aggregation operations. DataFrames often have hundreds of columns and millions of rows, and their full materialization in memory often pushes the limits of available memory on a machine [54, 57, 60]. By making remote memory available, AIFM can help data scientists interactively explore DataFrames without worrying about running out of memory.

We ported a popular open-source C++ DataFrame library [16] to AIFM's APIs. The primary data structure used in the library is an `std::vector` storing DataFrame columns and indexes, and we replaced this vector with the AIFM-enabled equivalent. In addition, we also added support for offloading key operations with low compute intensity but high memory access frequency to the remote side. We achieve this by offloading three operations using AIFM's remote device

		Offloaded Rem. Dev. Operations		
		Copy	Shuffle	Aggregate
DataFrame API	Filter	✓		
	Range extraction	✓		
	Add column/index	✓		
	Sort by column		✓	
	GroupBy		✓	✓

Table 1: DataFrame APIs (rows) and the offloaded operations they use via AIFM's remote device API (columns). Copy and Shuffle are memory-only operations, while Aggregate performs light remote-side computation.

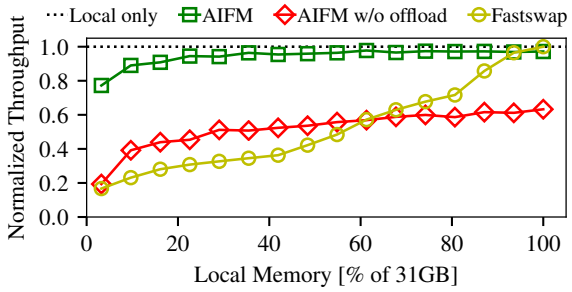


Figure 7: An AIFM-enabled DataFrame library [16] achieves 78–97% of in-memory throughput for a data analytics workload [53], outperforming Fastswap. Offloading operations with low compute intensity is crucial to AIFM's performance.

API (§4.2.4). The *Copy* and *Shuffle* operations copy a vector (*i.e.*, a DataFrame column), with shuffle also reordering rows by index positions in another column; *Aggregate* computes aggregate values (sums, averages, etc.). These three operations are used in five DataFrame API calls, including filters, column creation, sorts, and aggregations (Table 1). To achieve coverage sufficient to run the New York City taxi trip analysis workload [53], we modified 1,192 lines of code in the DataFrame library (which has 24.3k lines), and wrote 233 lines of remote device code. These modifications took one author about five days.

We benchmark our AIFM-enabled DataFrame with the Kaggle NYC taxi trip analysis workload [53], which explores trip dimensions including the number of passengers, trip durations, and distances, on the NYC taxi trip dataset [74] (16GB). The workload's full in-memory working set is 31GB. In the experiment, we vary the size of available local memory between 1GB and 31GB. We compare AIFM with Fastswap and a baseline with all data in local memory. In addition, we also investigate the impact of offloading on this workload, which consists of an operation with low compute intensity (*Aggregate* in Table 1) and some pure memory-copy operations (*Copy* and *Shuffle*). We would hope to find AIFM outperform Fastswap and come close to the local memory baseline.

Figure 7 shows the results. AIFM achieves 78% of in-memory throughput even with 1GB of local memory (3.2%) and exceeds 95% of ideal performance from about 20% (6GB) local memory. Fastswap, by contrast, achieves only 20% of in-memory performance at 1GB and only comes close to it once

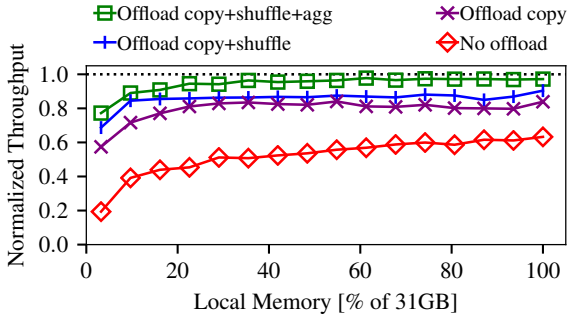


Figure 8: Performance gains from offloading the operations in Table 1. AIFM benefits most from offloading *Copy*, which increases throughput by 18–38%.

over 90% of the working set are in local memory. AIFM's high performance comes from avoiding Fastswap's page fault overheads, and from reducing expensive data movements over network by offloading operations with low compute intensity. Without offloading, AIFM outperforms Fastswap until 60% of the working set are local, as Fastswap incurs frequent minor faults. Beyond 60%, the fault rate in Fastswap drops sufficiently for most memory accesses to outperform AIFM's dereference-time overhead for low compute intensity operations (*e.g.*, memory copies). Offloading these operations to the remote side helps AIFM avoid this cost, while high compute-intensity operations amortize the dereference cost and happen locally. We also prototyped a batched API for AIFM that amortizes the dereference overhead across groups of vector elements when offloading is not possible, and found that it improves AIFM's throughput without offloading to 60–80% of in-memory throughput. We believe this could make a good future addition to AIFM's API to speed up low compute intensity operations if they must be performed locally.

Figure 8 breaks down the effect of offloading. Offloading *Copy* contributes the largest throughput gains (18%–38%); offloading *shuffle* contributes 2.9%–13%; and offloading *Aggregate* contributes 4.5%–12%. These results show that AIFM achieves high performance with small local memory for a real-world workload, and that AIFM's operation offloading is crucial to good performance when a workload includes operations with low compute intensity.

8.2 Data Structures

We pick two representative data structures—the hashtable and the array—from §6. We evaluate them in isolation, and explore the impact of prefetching, non-temporal local storage, and read/write amplification-reducing techniques.

8.2.1 Hashtable

Hash tables provide unordered maps that typically see random accesses, often with high temporal locality. A remoteable hash table should benefit from temporal caching of popular key-value (KV) pairs in local memory. Note that with AIFM, the caching policy is controlled by the data structure engi-

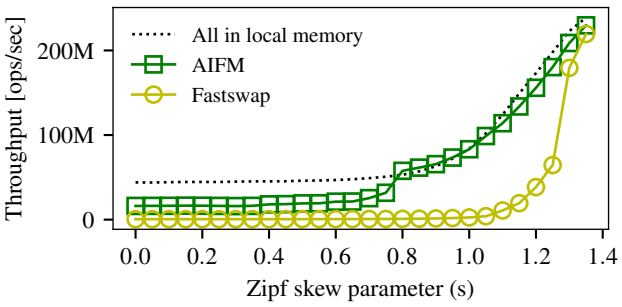
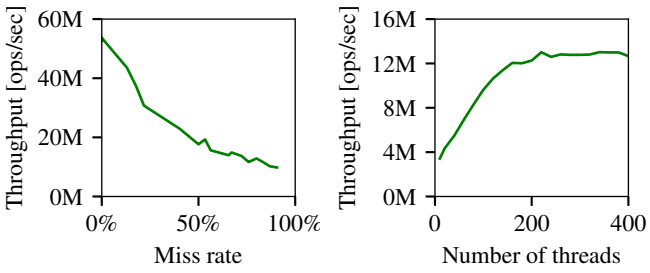


Figure 9: An AIFM hash table is competitive with local memory when the access distribution is skewed (Zipf factors ≥ 0.8), and outperforms a hashtable in Fastswap by up to 61× as Fastswap suffers from amplification and other overheads.

neer, while with Fastswap (or any swap-based far memory system), the caching policy is determined by the kernel page-reclamation policy, which in turn is based on page-granular hotness information.

Comparison. We evaluate the hashtable over Fastswap and AIFM with a memcached-style workload that issues `GET` requests, with keys sampled from a Zipf distribution whose parameter s we vary. Our key and value sizes are based on those reported for Facebook's USR memcached pool [8]. We load the hash table with 128M KV pairs (10GB total data), and compare performance to a baseline that keeps the entire hash table in local memory. Fastswap and AIFM instead allow a maximum of 5GB local data, split as follows. In Fastswap, the OS manages the both hashtable index (6GB) and value data (4GB) in swapable memory, with least recently used (LRU) [77] eviction at page granularity to decide on remote pages. In AIFM, we provision 3GB local memory region for index data and the other 2GB local memory region for value data; the runtime manages them separately. The hashtable's own object-granular CLOCK replacement algorithm guides AIFM's memory evacuator to pick KV pairs to evict to remote memory. In this experiment, we use a hashtable configured as an exclusive cache, *i.e.*, the evacuation handler removes local index entries for remote key-value pairs.

Figure 9 shows the throughput achieved as a function of the Zipf parameter s, ranging from near-uniform at zero to highly skewed at $s = 1.35$. AIFM achieves about 17M operations/second at low skew ($\approx 60\%$ miss rate at $s = 0$), about one third of the 53M operations/second that a fully-local hash table achieves. As skew increases and the miss rate drops, AIFM comes closer to local-only performance: for example, at $s = 0.8$ (1% miss rate), it reaches 57M operations/second; and from $s = 0.8$, it matches the performance of the local-only hashtable. Fastswap, by contrast, sees a throughput of 0.54M operations/second at $s = 0$ (30× less than AIFM) and only matches the local-only baseline beyond $s = 1.3$. At $s = 0.8$, AIFM has its largest advantage over Fastswap (61×).



(a) `GET` throughput as a function of the miss rate. **(b)** `GET` throughput as a function of thread count (80% miss rate).

Figure 10: AIFM hash table microbenchmarks.

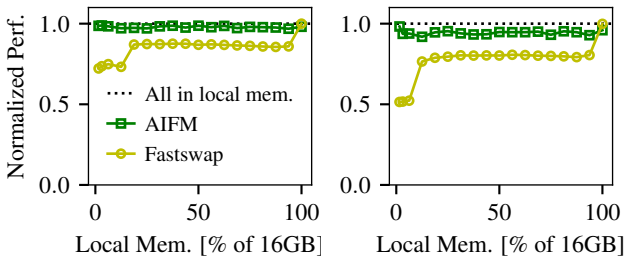
This difference comes from three factors against Fastswap: (*i*) amplification due to page-granular swapping, (*ii*) lack of per-KV pair hotness information, and (*iii*) the overheads of kernel paging. Since a page contains 128 key-value pairs, page-granular swapping incurs up to 128× read and write amplification. This amplification increases the network bandwidth required and pollutes the local memory, increasing Fastswap's miss rate with identical memory available. For example, at $s = 1.25$, Fastswap still uses 140MB/s of network bandwidth, while AIFM's bandwidth use rapidly drops beyond $s = 0.8$. Fastswap also cannot swap out only cold key-value pairs, as a page contains entries with varying hotness, but the kernel tracks access only at page granularity. Finally, Fastswap incurs the cost of kernel crossings, page faults, identifying and reclaiming victim pages (38% of cycles at $s = 0.8$) and wasted cycles waiting for I/O (49%). AIFM's overheads are limited to running the evacuator (0.8% of cycles at $s = 0.8$), TCP stack overheads (1.7%), and thread scheduler overhead (14%).

Microbenchmarks. Figure 10a shows how hash table performs at different miss rates when requests are uniformly, rather than Zipf-distributed. It achieves a best-case throughput of 53M requests/second, reduced to 10M requests/second when it is close to 100% miss rate. Figure 10b measures, for the same uniform distribution and an 80% miss rate, the throughput AIFM achieves with an increasing number of application threads. Up to 160 threads, AIFM extracts more throughput by scheduling additional requests while it waits for requests to complete.

8.2.2 Array

Depending on the access pattern, an array may benefit from caching (for random access with temporal locality), prefetching (for sequential access), and non-temporal storage (if there is no temporal locality).

We evaluate our array with the Snappy library [30]. The benchmark performs in-memory compression/decompression by reading input files from a `RemArray` and writing output files to another `RemArray`. For benchmarking compression, we use 16 input files of 1GB each. For decompression, we use



(a) Compression. **(b)** Decompression.

Figure 11: AIFM achieves nearly identical performance to local memory when compressing/decompressing an array with Snappy [30] (sequential access), and outperforms Fastswap.

30 input files of 0.5GB each. The compression ratio is around 2. Both operations perform streaming, sequential access to the array and never revisit any object. We compare Fastswap and an ideal, completely local in-memory baseline.

AIFM's array prefetcher captures application semantics through the array access APIs and performs prefetching entirely in user space. OS-based paging systems, by contrast, must rely on page faults (major faults for unprefetched pages and minor faults for prefetched pages) to pass application semantics, which imposes high overheads. For each system, we measure performance with different amounts of local memory available (for Fastswap, we restrict memory via cgroups; for AIFM, we set the local memory size). A good result would avoid AIFM's slow path, as every far pointer dereference would find local data already.

Figure 11 shows the results. We see that AIFM achieves performance close to the in-memory baseline, independent of the local memory size, while Fastswap's performance depends on local memory size and only matches AIFM when nearly all memory is local. This demonstrates the benefit of AIFM's non-temporal access and prefetching.

8.3 Design Drill-Down

We now evaluate specific aspects of the AIFM design using microbenchmarks.

8.3.1 Fast/Slow Path Costs

AIFM seeks to provide access to local objects with latency close to normal memory access. This means that AIFM's remoteable pointer must minimize overheads on the "fast path", when no remote memory access is required.

We measured the hot path latency of dereferencing a `RemUniquePtr` and compared it to the latency for dereferencing a C++ `unique_ptr`, both when the pointer and data pointed to are cached and uncached. Figure 12a shows that AIFM offers comparable latency to an ordinary C++ smart pointer. For an object in L1 cache, AIFM has a 4× latency overhead: four micro-ops *vs.* a single pointer dereference operation. In practice, modern CPU's instruction-level parallelism

90th percentile latency [cycles]	*read*	*write*
C++ `unique_ptr` (uncached)	570	408
AIFM object (uncached)	489	309

(a) Hot path (local object).

90th percentile latency [cycles]		*read*	*write*
Fastswap total		23,712	26,382
... of which	RDMA transfer (4KB)	16,521	16,521
	Overheads	7,191	9,861
AIFM total (64B object)		18,582	18,369
... of which	TCP transfer (64B)	17,694	17,673
	Overheads	888	696
AIFM total (4KB object)		27,183	27,279
... of which	TCP transfer (4KB)	26,055	26,121
	Overheads	1,128	1,158

(b) Cold path (remote object).

Figure 12: AIFM is competitive with an ordinary pointer dereference, and it has lower overheads than Fastswap.

hides some of this latency, and we observe a 2× throughput overhead for L1 hits.

We also measured AIFM's cold path latency, and compared it to Fastswap's. Fastswap always fetches at least 4KB from the remote server, but its RDMA backend is faster than AIFM's TCP backend. This might amortize some of the overheads associated with page-granular far memory that Fastswap suffers from. A good result would show AIFM with comparable latency to Fastswap for large objects (4KB), and lower latency for small objects (64B).

Figure 12b shows the results. While Fastswap's raw data transfers are indeed faster than AIFM's, AIFM achieves lower latency for cache-line-sized (64B) objects due to its 10× lower overheads. For 4KB objects, AIFM is close to Fastswap, but has 10% higher latency on reads; AIFM with an RDMA backend would come closer. In addition, AIFM can productively use its wait cycles, which yields a 1.8–6.8× throughput increase over Fastswap (Figure 1).

8.3.2 Operating Point

AIFM is designed for applications that perform some compute for each remoteable data structure access, as this compute allows AIFM to hide the latency of far memory by prefetching. But if an application has a huge amount of compute per data structure access, AIFM will offer limited benefit over page-granular approaches like Fastswap, despite their overheads. We ran a sensitivity analysis with a synthetic application that spins for a configurable amount of time in between sequential accesses into a remoteable array. This should allow AIFM's prefetcher to run ahead and load successive elements before they are dereferenced. We compare to Fastswap, which we configure with the maximum prefetching window (32 pages).

Figure 13 shows the results, normalized to the benchmark runtime against a purely in-memory array. AIFM becomes competitive with local memory access from about 1.2µs of compute between array accesses. Fastswap's overheads amor-

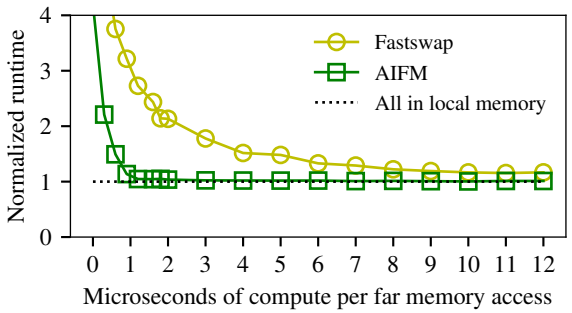


Figure 13: AIFM becomes competitive with local memory access at around 1.2µs of compute per sequential far memory access (4KB object) in a microbenchmark, while kernel-based swapping mechanisms require higher compute ratios (ca. 50µs per memory access; not shown) to compete.

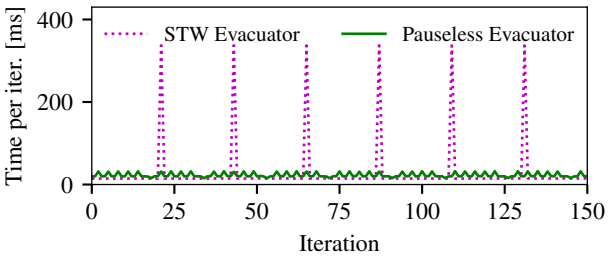


Figure 14: Pauseless evacuation is essential for low latency accesses: a stop-the-world (STW) evacuator frequently encounters 10× higher latency as it swaps out objects.

tize more slowly—its line converges with AIFM's around 50µs of compute per array access. This demonstrates that AIFM supports efficient remote memory in a wider range of applications than page-granular approaches like Fastswap.

8.3.3 Memory Evacuator

We evaluate two key aspects of AIFM's memory evacuator design: the choice to never pause mutator threads (§5.3) and the thread scheduler co-design (§5.4).

Pauseless Evacuation. In this experiment, we run 10 mutator threads (the number of physical CPU cores in our machine) that keep entering the dereference scope, dereferencing and marking dirty 4MB of data each time. Therefore, the runtime periodically triggers memory evacuation. We compare AIFM's pauseless evacuator design to a stop-the-world memory evacuator, and measure the latency per mutator iteration (4MB write). Figure 14 shows that a stop-the-world evacuator design causes periodic mutator latency spikes up to 340ms. By contrast, AIFM's pauseless evacuator consistently runs an iteration in about 25ms. (The tiny spikes of the pauseless line are mainly caused by hyperthread and cache contention between evacuators and mutators.) This confirms that a pauseless evacuator is essential to consistent application performance.

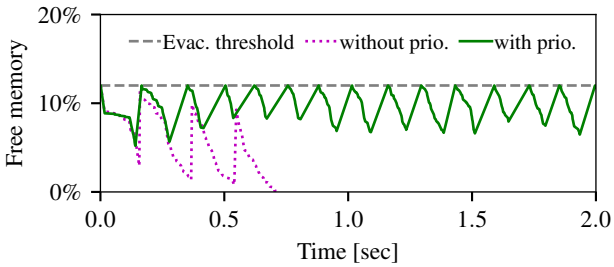


Figure 15: Thread prioritization in the runtime is essential to ensure that evacuation always succeeds. 12% free memory is the threshold for AIFM to trigger evacuation.

Thread Scheduler Co-design. In this experiment, we run 100 mutator threads that each iterates to read 1MB of data from a remoteable array and perform 20ms of computation. We run AIFM with the scheduler's thread prioritization (§5.4) enabled and disabled, and measure the free local memory over time. For a responsive system, local memory should never run out entirely, and the evacuator should be able to free memory fast enough to keep up with the mutators.

Figure 15 shows that the runtime without prioritization fails to keep up and runs out of memory after around 0.7 seconds. AIFM's prioritizing scheduler, on the other hand, ensures that sufficient memory remains available. This illustrates that the benefit of co-locating thread scheduler and memory evacuator in a user-space runtime.

9 Conclusion

We presented Application-Integrated Far Memory (AIFM), a new approach to extending a server's available RAM with high-performance remote memory. Unlike prior, kernel-based, page-granular approaches, AIFM integrates far memory with application data structures, allowing for fine-grained partial remoting of data structures without amplification or high overheads. AIFM is based on four key components: (*i*) the remote pointer abstraction; (*ii*) the pauseless memory evacuator; (*iii*) the data structure APIs with rich semantics; (*iv*) and the remote device abstraction. All parts work together to deliver high performance and convenient APIs for application developers and data structure engineers.

Our experiments show that AIFM delivers performance close to, or on par with, local DRAM at operating points that prior far memory systems could not efficiently support.

AIFM is available as open-source software at https://github.com/aifm-sys/aifm.

Acknowledgements

We thank our shepherd Emmett Witchel, the anonymous reviewers, and members of the MIT PDOS group for their helpful feedback. We appreciate Cloudlab [25] for providing the experiment platform used. This work was supported in part by a Facebook Research Award and a Google Faculty Award. Zhenyuan Ruan was supported by an MIT Robert J. Shillman Fund Fellowship.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. "FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. "Remote regions: a simple abstraction for remote memory". In: *USENIX Annual Technical Conference (ATC)*. 2018.

[3] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. "Remote Memory in the Age of Fast Networks". In: *ACM Symposium on Cloud Computing (SoCC)*. 2017.

[4] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. "Designing far memory data structures: Think outside the box". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2019.

[5] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. "Write-Rationing Garbage Collection for Hybrid Memories". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2018.

[6] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. "Can Far Memory Improve Job Throughput?" In: *European Conference on Computer Systems (EuroSys)*. 2020.

[7] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. "Treadmarks: Shared memory computing on networks of workstations". In: *Computer* 29.2 (1996).

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload Analysis of a Large-Scale Key-Value Store". In: *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 2012.

[9] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. "Attack of the killer microseconds". In: *Communications of the ACM* 60.4 (2017).

[10] John K. Bennett, John B. Carter, and Willy Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-specific Memory Coherence". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1990.

[11] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. "The End of Slow Networks: It's Time for a Redesign". In: *Proceedings of the VLDB Endowment* 9.7 (2016).

[12] Andrew D. Birrell and Bruce Jay Nelson. "Implementing Remote Procedure Calls". In: *ACM Transactions on Computer Systems (TOCS)* 2.1 (1984).

[13] Hans-Juergen Boehm and Mark Weiser. "Garbage collection in an uncooperative environment". In: *Software: Practice and Experience* 18.9 (1988).

[14] Michael D. Bond and Kathryn S. McKinley. "Tolerating Memory Leaks". In: *ACM SIGPLAN Notices* 43.10 (2008).

[15] Benjamin Brock, Aydın Buluç, and Katherine Yelick. "BCL: A cross-platform distributed container library". In: *International Conference on Parallel Processing (ICPP)*. 2019.

[16] "C++ DataFrame for statistical, Financial, and ML analysis". In: `https://github.com/hosseinmoein/DataFrame` (last accessed on 10/15/2020).

[17] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. "Project PBerry: FPGA Acceleration for Remote Memory". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2019.

[18] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A cloud-scale acceleration architecture". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016.

[19] Youmin Chen, Youyou Lu, and Jiwu Shu. "Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing". In: *European Conference on Computer Systems (EuroSys)*. 2019.

[20] Cliff Click, Gil Tene, and Michael Wolf. "The pauseless GC algorithm". In: *ACM/USENIX international conference on Virtual execution environments (VEE)*. 2005.

[21] Douglas Comer and Jim Griffioen. "A New Design for Distributed Systems: The Remote Memory Model". In: *Summer USENIX Conference*. 1990.

[22] "Crypto++ Library 8.2". In: https://www.cryptopp.com/ (last accessed on 10/15/2020).

[23] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. "Garbage-First Garbage Collection". In: *International Symposium on Memory Management (ISMM)*. 2004.

[24] Fred Douglis. "The compression cache: Using on-line compression to extend physical memory". In: *Winter USENIX Conference*. 1993.

[25] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. "The Design and Operation of CloudLab". In: *USENIX Annual Technical Conference (ATC)*. 2019.

[26] "Facebook Folly RCU Library". In: https://github.com/facebook/folly/blob/master/folly/synchronization/Rcu.h (last accessed on 10/15/2020).

[27] Michail D. Flouris and Evangelos P. Markatos. "The Network RamDisk: Using Remote Memory on Heterogeneous NOWs". In: *Cluster Computing* 2.4 (1999).

[28] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "Network Requirements for Resource Disaggregation". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[29] "Gen-Z: hardware architecture for disaggregated memory". In: https://genzconsortium.org (last accessed on 10/15/2020).

[30] "Google's fast compressor/decompressor". In: https://github.com/google/snappy (last accessed on 10/15/2020).

[31] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. "Efficient Memory Disaggregation with Infiniswap". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[32] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. "RDMA over Commodity Ethernet at Scale". In: *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 2016.

[33] "HPE Powers Up The Machine Architecture". In: https://www.nextplatform.com/2017/01/09/hpe-powers-machine-architecture (last accessed on 10/15/2020).

[34] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. "The Garbage Collection Advantage: Improving Program Locality". In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2004.

[35] "Intel 64 and IA-32 Architectures Developer's Manual: Vol. 1". In: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html (last accessed on 10/15/2020).

[36] "Java SE documentation. Chapter 6: The Parallel Collector". In: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html (last accessed on 10/15/2020).

[37] Anuj Kalia, Michael Kaminsky, and David Andersen. "Datacenter RPCs can be General and Fast". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2019.

[38] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-value Services". In: *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 2014.

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *USENIX Annual Technical Conference (ATC)*. 2016.

[40] Samir Koussih, Anurag Acharya, and Sanjeev Setia. "Dodo: A User-level System for Exploiting Idle Memory in Workstation Clusters". In: *IEEE International Symposium on High Performance Distributed Computing (HPDC)*. 1998.

[41] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. "Software-Defined Far Memory in Warehouse-Scale Computers". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[42] Butler W Lampson and David D Redell. "Experience with processes and monitors in Mesa". In: *Communications of the ACM* 23.2 (1980).

[43] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[44] Kai Li and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems". In: 7.4 (1989).

[45] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. "Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device". In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2005.

[46] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. "Imbalance in the cloud: An analysis on alibaba cluster trace". In: *IEEE International Conference on Big Data (Big Data)*. IEEE. 2017.

[47] "Mellanox Innova-2 Flex Open Programmable SmartNIC". In: https://www.mellanox.com/products/smartnics/innova-2-flex (last accessed on 10/15/2020).

[48] Feeley Michael J, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. "Implementing Global Memory Management in a Workstation Cluster". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 1995.

[49] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store". In: *USENIX Annual Technical Conference (ATC)*. 2013.

[50] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. "Latency-tolerant Software Distributed Shared Memory". In: *USENIX Annual Technical Conference (ATC)*. 2015.

[51] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. "Storm: a fast transactional dataplane for remote data structures". In: *ACM International Conference on Systems and Storage (SYSTOR)*. 2019.

[52] "NVIDIA Mellanox BlueField DPU". In: https://www.mellanox.com/products/bluefield-overview (last accessed on 10/15/2020).

[53] "NYC Taxi Trips - Exploratory Data Analysis". In: https://www.kaggle.com/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook (last accessed on 10/15/2020).

[54] "Opening a 20GB file for analysis with pandas". In: https://datascience.stackexchange.com/questions/27767/opening-a-20gb-file-for-analysis-with-pandas/ (last accessed on 10/15/2020).

[55] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2019.

[56] "pandas - Python Data Analysis Library". In: https://pandas.pydata.org/ (last accessed on 10/15/2020).

[57] "Pandas: Scaling to large datasets". In: https://pandas.pydata.org/pandas-docs/stable/user_guide/scale.html (last accessed on 10/15/2020).

[58] Nathan Pemberton. "Exploring the disaggregated memory interface design space". In: *Workshop on Resource Disaggregation (WORD)*. 2019.

[59] "Persistent Memory Development Kit". In: https://pmem.io/pmdk/ (last accessed on 10/15/2020).

[60] "Quora: Is anyone successful in using Python Pandas while dealing with millions of rows or more than a billion?" In: https://www.quora.com/Is-anyone-successful-in-using-Python-Pandas-while-dealing-with-millions-of-rows-or-more-than-a-billion-If-not-what-else-did-you-do (last accessed on 10/15/2020).

[61] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. "PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[62] Andy Rudoff. "Persistent memory programming". In: *Login: The Usenix Magazine* 42 (2017).

[63] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. "Log-structured Memory for DRAM-based Storage". In: *USENIX Conference on File and Storage Technologies (FAST)*. 2014.

[64] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. "Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1996.

[65] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. "Fine-grain Access Control for Distributed Shared Memory". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1994.

[66] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[67] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. "Distributed shared persistent memory". In: *ACM Symposium on Cloud Computing (SoCC)*. 2017.

[68] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. "StRoM: smart remote memory". In: *European Conference on Computer Systems (EuroSys)*. 2020.

[69] "std::weak_ptr". In: https://en.cppreference.com/w/cpp/memory/weak_ptr (last accessed on 10/15/2020).

[70] "Stingray SmartNIC Adapters and IC". In: https://www.broadcom.com/products/ethernet-connectivity/smartnic (last accessed on 10/15/2020).

[71] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. "DaRPC: Data Center RPC". In: *ACM Symposium on Cloud Computing (SoCC)*. 2014.

[72] Andrew S Tanenbaum and Herbert Bos. *Modern Operating Systems*. 2015.

[73] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: The next Generation". In: *European Conference on Computer Systems (EuroSys)*. 2020.

[74] "TLC Trip Record Data". In: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page (last accessed on 10/15/2020).

[75] Po-An Tsai and Daniel Sanchez. "Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[76] Shin-Yeh Tsai and Yiying Zhang. "LITE Kernel RDMA Support for Datacenter Applications". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[77] "Understanding the Linux Virtual Memory Manager, Chapter 10 Page Frame Reclamation". In: https://www.kernel.org/doc/gorman/html/understand/understand013.html (last accessed on 10/15/2020).

[78] "Validating References with Lifetimes". In: https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html (last accessed on 10/15/2020).

[79] Chenxi Wang, Ting Cao, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. "Efficient Management for Hybrid Memory in Managed Language Runtime". In: *Network and Parallel Computing (NPC)*. Edited by Guang R. Gao, Depei Qian, Xinbo Gao, Barbara Chapman, and Wenguang Chen. 2016.

[80] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. "Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019.

[81] Paul Wilson. "Operating System Support for Small Objects". In: *International Workshop on Object Orientation in Operating Systems (IWOOOS)*. 1991.

[82] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. "The Case for Compressed Caching in Virtual Memory Systems". In: *USENIX Annual Technical Conference (ATC)*. 1999.



Performance-Optimal Read-Only Transactions

Haonan Lu*, Siddhartha Sen†, Wyatt Lloyd*
*Princeton University, †Microsoft Research

Abstract

Read-only transactions are critical for consistently reading data spread across a distributed storage system but have worse performance than simple, non-transactional reads. We identify three properties of simple reads that are necessary for read-only transactions to be performance-optimal, i.e., come as close as possible to simple reads. We demonstrate a fundamental tradeoff in the design of read-only transactions by proving that performance optimality is impossible to achieve with strict serializability, the strongest consistency.

Guided by this result, we present PORT, a performance-optimal design with the strongest consistency to date. Central to PORT are version clocks, a specialized logical clock that concisely captures the necessary ordering constraints. We show the generality of PORT with two applications. Scylla-PORT provides process-ordered serializability with simple writes and shows performance comparable to its non-transactional base system. Eiger-PORT provides causal consistency with write transactions and significantly improves the performance of its transactional base system.

1 Introduction

Large-scale web services are built on distributed storage systems. Sharding data across machines enables distributed storage systems to scale capacity and throughput. Sharding, however, complicates building correct applications because read requests sent to different shards may arrive at different times and thus return an inconsistent view of the data.

Consistently interacting with data in a distributed storage system thus requires transactional *isolation*, which unifies the view of data across shards. While general transactions provide isolation for reading and writing across shards, this paper focuses on *read-only transactions* that only read data. Read-only transactions are prevalent: they are used in systems without general transactions [4, 14, 31, 32, 34] and, even for systems with general transactions, they are often implemented with a specialized algorithm [10, 11, 34, 37, 38, 39, 51]. Read-only transactions are practically important because reads dominate real-world workloads: Facebook reported 99.8% reads for TAO [8] and Google reported three orders of magnitude more reads than general transactions for the ads workload (F1) that runs on Spanner [10]. They are also theoretically important because they provide a lower bound for other classes of transactions: anything impossible for read-only transactions is also impossible for any class of transactions that includes reads.

The dominance of reads in real-world workloads makes their performance the primary determinant of end-user latency and overall system throughput. Unfortunately, read-only transactions perform worse than simple, non-transactional reads due to the coordination required to present a consistent view across shards. Whether a view is consistent is determined by a system's *consistency model*: stronger consistency provides an abstraction closer to a single-threaded environment, greatly simplifying application code [33]. Thus, ideal read-only transactions would provide the strongest consistency *and* have optimal performance.

What is the "optimal" performance? Although recent work has studied optimality through the lens of latency [34], it did not consider throughput, which adds a fundamentally new dimension to this question. In this paper, we formalize the notion of optimality for read-only transactions and use it to explore the tradeoff between their consistency and performance. We posit that optimality should be defined by the algorithmic properties of simple reads that comprise a read-only transaction. *Simple reads* do not provide transactional isolation and thus capture the minimum work required to read data in a distributed storage system: **O**ne round of **N**on-blocking communication with a **C**onstant amount of metadata. As we elaborate in §3, these algorithmic properties (N, O, and C) precisely capture the additional coordination incurred by read-only transactions to present a consistent view. Thus, we define *performance-optimal* read-only transactions to be those with the same NOC properties as simple reads.

Our main theoretical result is that performance optimality is impossible in a system that provides **S**trict serializability—the strongest type of consistency. Specifically, our NOCS Theorem states that no read-only transaction algorithm can be performance optimal and provide strict serializability. This result holds even in systems that only support non-transactional writes, and thus applies to systems with and without more general types of transactions. It shows there is a fundamental choice in the design of distributed storage systems: they can either provide the strongest consistency or the best performance for read-only transactions, not both.

Guided by our impossibility result, we present the PORT design, which enables performance-optimal read-only transactions with the strongest consistency to date: process-ordered serializability. Previous performance-optimal transactions only provided relatively weak consistency (§5.1). PORT provides performance-optimal read-only transactions without harming either the latency or throughput of writes. The main mechanism enabling our design is a new special-

ized logical clock, called *version clocks*, that concisely capture the ordering constraints imposed by process-ordered serializability on read and write operations. PORT uses version clocks to tightly co-design its components. Version clock values index its multi-versioning framework, control what read-only transactions see, and control where writes are applied. They also enable optimizations that avoid the work of applying some concurrent writes (*write omission*) and limit the staleness of reads (*data freshness*).

We use the PORT design with the write omission and data freshness optimizations to build a new storage system, Scylla-PORT, that adds performance-optimal read-only transactions to ScyllaDB [47] while providing process-ordered serializability. As a single-versioned, non-transactional system, ScyllaDB provides a clean slate for implementing PORT and allows us to quantify the overhead of our performance-optimal read-only transactions relative to simple reads. ScyllaDB's simple reads are a challenging baseline as the system is aggressively engineered for high performance, including core-level sharding and custom lock-free data structures. Our evaluation shows that PORT's read-only transactions introduce low overhead, achieving throughput and latency within 3% of ScyllaDB on most of the workloads we test, and within 8% in the worst case. Our evaluation also compares PORT to a variant of OCC that is optimized for read-only transactions. PORT significantly outperforms OCC with at least double the throughput and at most half the latency because Scylla-PORT always finishes in one round while OCC's best case is two rounds.

We also applied PORT with data freshness optimizations to Eiger [32] to make its read-only transactions performance optimal while preserving the system's causal consistency and write transactions. Eiger is a challenging baseline because it can complete read-only transactions in a single round. Our evaluation shows that Eiger-PORT significantly improves performance with throughput up to 3× higher and latency up to 60% lower than Eiger. These improvements do come with some staleness relative to strongly consistent systems, but our data freshness optimizations keep the staleness low.

In summary, this work makes the following contributions:

- A fundamental understanding of the tradeoff between performance and consistency for read-only transactions. This includes a precise definition of performance optimality (§3) and the NOCS Theorem that proves optimality is impossible with strict serializability (§4).
- The PORT design that achieves performance-optimal read-only transactions with the strongest consistency to date by leveraging version clocks, a new type of logical clock that concisely captures the necessary ordering constraints (§6).
- The implementation and evaluation of two new systems based on the PORT design. Scylla-PORT is a clean-slate application of PORT to a non-transactional system, ScyllaDB (§7). Eiger-PORT makes the read-only transaction algorithm of Eiger performance optimal (§8, §9).

2 Background

Web service architecture. Web services are typically built using two tiers of machines: a stateless frontend tier and a stateful storage tier. The frontends handle end user requests by executing application logic that generates sub-requests to read or write data in the storage tier. We refer to the frontends as *clients* and the storage machines as *servers*, as is common. Web services are often replicated across multiple datacenters. For simplicity, we focus on a single datacenter setting, but our results also apply to multi-datacenter settings.

Read-only transactions. Read-only transactions provide a consistent, unified view of data spread across servers in a storage tier. They consist of one or more logical rounds of simple read requests issued in parallel to the servers, which collectively return a view satisfying the consistency model of the system. *One-shot transactions* [23] know the data locations of all reads prior to the transaction start. In contrast, *multi-shot transactions* may include key dependencies, where the data read in one shot determines what data to read in later shots. We study one-shot transactions for simplicity, because they are common, and because what is impossible for them is also necessarily impossible for multi-shot transactions. The NOCS Theorem thus also applies to multi-shot transactions. The PORT design for read-only transactions can be easily extended to support multi-shot transactions.

3 Performance-Optimal Read Transactions

This section explains the challenges of reasoning about performance, the rationale of our approach, and the set of algorithmic properties that define optimal performance.

3.1 Reasoning About Performance

The key challenges to reasoning about performance are identifying the fundamental overhead of read-only transactions and modeling it in a way that connects with practical designs.

Capturing the fundamental overhead. As a layer built upon simple reads, the performance of a read-only transaction is impacted by both the engineering factors in executing simple reads and the algorithmic properties of coordinating simple reads to find a consistent view. Engineering factors, such as load balancing, batching, and networking, equally affect simple reads and the read-only transactions built on them. In contrast, the algorithmic properties, such as rounds of communication, only affect read-only transactions. For instance, a read-only transaction protocol that requires multiple round trips incurs overhead due to those extra rounds of messages, while the read requests in each round are engineered the same as simple reads.

Thus, this work focuses on the algorithmic properties that capture the fundamental overhead of read-only transactions. These properties capture the additional overhead to coordinate a consistent view and are orthogonal to underlying engineering factors. More specifically, we answer the question,

"given a system, how low can we make the performance overhead of read-only transactions relative to the system's simple reads?"

Being useful in practice. Our goal is to model optimal performance in a way that is both *theoretically insightful* and *practically useful*. Theoretical insights help clarify fundamental tradeoffs between performance and guarantees. Practically useful guidance helps us design better systems. Our NOCS Theorem (§4.1) and properties yield theoretical insights that lead to a better design, PORT (§6), that achieves better performance in practice. This shows that our modeling is practically useful (§5).

3.2 Approach Overview

To reason about optimal performance in a practically useful way, we examine the mechanisms used in existing systems to coordinate a consistent view across shards. These coordination mechanisms include blocking, extra messages, and metadata. Some systems *block* read operations until a consistent view is ready—e.g., systems that use two-phase locking. Almost all systems use *extra messages* to determine a consistent view, such as multiple round trips on the critical path of reads—e.g., OCC [24]—or approaches that asynchronously coordinate a consistent view—e.g., COPS-SNOW [34], GentleRain [15], Cure [3]. Finally, all systems we are aware of use *metadata* to help compute a consistent view for read-only transactions to return—e.g., timestamps, transaction ids. Figure 9 in Section 10 shows representative systems that use these mechanisms.

These coordination mechanisms cause read-only transactions to have worse performance than simple reads, as they consume additional system resources. Therefore, we define performance-optimal read-only transactions to be those that require the least amount of each coordination mechanism, making their performance closest to that of simple reads.

3.3 NOC: Optimal Performance

We now explain the NOC properties, which we use to define optimal performance for read-only transactions.

N: Non-blocking. A read-only transaction algorithm is *non-blocking* if servers process each read request without waiting for any external event, such as a lock to become available, a message to arrive, or a timer to expire.

Blocking for a read request increases the latency of the read-only transaction: the more time spent blocking, the longer the transaction takes to complete. It also decreases throughput due to the overhead of context switches. In practice, blocking can incur more serious performance issues, e.g., CPU underutilization and deadlocks, which are increasingly pronounced in modern services [44, 52].

O: One-round communication. A read-only transaction algorithm has *one-round communication* if it uses exactly one parallel round of on-path messages and does not have any off-path messages. This matches the messages of simple reads: the client sends a single request to each server holding relevant data, and each server sends a single response back. It excludes algorithms that use extra messages, such as those that require multiple rounds of on-path communication, e.g., to abort/retry. It also disallows coordinating through *off-path messages*, i.e., messages that are necessary for the read-only transactions but lie off the critical path of reads.

A message is an off-path message for read-only transactions if its removal affects *only* the correctness of read-only transactions. For example, COPS-SNOW [34] adds extra messages to writes. These messages are used for read-only transactions to find a consistent snapshot and are not necessary for processing writes. Because only the correctness of read-only transactions is affected if these messages are removed, they are off-path messages.

Additional rounds of on-path messages increase the latency of read-only transactions. Both extra on-path and off-path messages decrease system throughput because transmitting and processing them consume network and CPU resources that could otherwise be used to service requests.

C: Constant metadata. Metadata is the information required by a read-only transaction algorithm to coordinate consistent values. It is information a server needs to find the specific version of the data that will produce a consistent cross-shard view across reads in the same transaction. Examples of metadata include timestamps [2, 10], transaction ids [34, 41], and identifiers of participating servers [5].

A read-only transaction algorithm has *constant metadata* if the amount of metadata required to process each of its read requests is constant, i.e., it does not increase with the size of the system, the size of the transaction, or the number of concurrent operations. An example of constant metadata is one timestamp per read request for snapshot reads in Spanner [10]. An example of non-constant metadata is COPS-SNOW [34], which requires information about *many* concurrent read-only transactions to process each read request.

Transmitting and/or processing extra metadata consumes more resources, increasing latency and decreasing throughput. Its negative impact on performance has been reported in recent work [13, 14, 15]. We use Big-O notation, i.e., "constant," to capture the algorithmic complexity of metadata required for coordination. In practice, system designers should aim for as low a constant as possible. We realize this in our PORT design, which uses a single integer per read request.

Performance optimality. We deem an algorithm *performance optimal* if it satisfies the N+O+C properties because they capture the least coordination overhead and thus enable performance as close as possible to simple reads.

4 The NOCS Theorem

An ideal system would have performance-optimal read-only transactions that provide the strongest consistency. Our NOCS Theorem proves this ideal is impossible.

S: Strict serializability. Strict serializability is the strongest form of consistency, equivalent to linearizability [22] with the addition of transactional isolation. It requires that there exists a legal total order of transactions that respects the real-time order between transactions [42]. A *legal total order* ensures that the results of transactions are equivalent to a single entity processing them one by one. The *real-time order* ensures that if transaction T_2 starts after transaction T_1 ends, then T_1 must appear before T_2 in the total order. If T_1 and T_2 have overlapping lifetimes, then they are concurrent and can be placed in either order. Strict serializability gives application programmers the powerful abstraction of programming in a single-threaded, transactionally isolated environment.

4.1 NOCS is Impossible

Our main result is that performance-optimal read-only transactions (N+O+C) cannot provide strict serializability (S). This section present a condensed version of the proof. The full proof appears in our accompanying technical report [35].

The NOCS Theorem. *No read-only transaction algorithm satisfies all NOCS properties.*

System model. We model a distributed system as a set of processes that communicate by sending and receiving messages. This model is similar to that used in FLP [17]. A set of client processes (clients) issue requests to server processes (servers) that store the data. Processes are modeled as deterministic automata: in each atomic step, they may receive a message, perform deterministic local computation, and send one or more messages to other processes.

A transaction (operation) starts when a client sends the request messages to servers and ends when the client receives the last necessary server response. Two transactions (operations) are concurrent if their lifetimes overlap, i.e., neither begins after the other ends. If concurrent transactions (operations) access the same data item, then they conflict.

Assumptions. We make the following assumptions:

(A-0) There are ≥ 2 servers and ≥ 2 clients. Otherwise, optimal performance and strict serializability are trivial. All reads and writes eventually complete.

(A-1) The network and processors are reliable. Every message is eventually delivered and processed by the destination process. Processes are correct and never crash. By proving our impossibility result under these favorable conditions, it will necessarily hold when the system can fail.

(A-2) The network is either asynchronous [20], i.e., messages can be arbitrarily delayed, or partially synchronous [16], i.e., physical clocks ensure bounded delays.

Proof intuition. Due to network asynchrony, it is always possible for a read-only transaction to conflict with write operations and other concurrent read-only transactions. These requests occupy *an unstable region* in the system's history, where conflicts are possible and a total order has not yet been established. In contrast, the *stable region* is the part of history that precedes the unstable region, where all writes have committed and system states are finalized. Reading in the stable region is easy as there are no conflicting writes. However, we show that the real-time order requirement of S requires read-only transactions that are N+O to interact with the most recent writes in the unstable region (Lemma 1). Doing this while ensuring a legal total order requires transferring metadata between the servers (Lemma 2), either proactively through read requests or through the write protocol. By extending this construction, we show that processing a set of read-only transactions requires metadata that is asymptotically larger than the total size of the transactions, regardless of how the metadata is transferred (Lemma 3). This violates C, proving the theorem.

Proof. Suppose the system has two servers, S_1 and S_2, and multiple clients. Let `ALG` be any read-only transaction algorithm that satisfies N+O+S. Let $R = \{r_1, r_2\}$ be a read-only transaction that executes `ALG`, issued by client C_R. Let w_1 and w_2 be simple write requests issued by client $C_w \neq C_R$, where $w_1 \rightarrow w_2$ in real-time, i.e., w_2 is sent after the response for w_1 is received. We place no restrictions on the write protocol (beyond assumption A-0). Consider the execution e_1:

S_1 : r_1, w_1
S_2 : w_2, r_2

Suppose there is no metadata in the system, i.e., no information for coordinating consistent values between requests.

Lemma 1. *Without metadata, a read-only transaction that is N+O+S must observe any write that precedes it at a server.*

Proof Summary. Without metadata, S_2 cannot distinguish between an execution where w_2 and R are concurrent and one with $w_2 \rightarrow R$ in real-time. The latter requires $r_2 \in R$ to observe w_2 to satisfy S's real-time order. ■

Lemma 2. *Processing e_1 while satisfying N+O+S requires dependency $R \rightarrow w_1$ to be transferred from S_1 to S_2.*

Proof Summary. Lemma 1 states that, without metadata, r_2 must observe w_2, implying $w_2 \rightarrow R$. But r_1 must be processed before w_1 to satisfy N+O, implying $R \rightarrow w_1$. Since $w_1 \rightarrow w_2$ by construction, this creates a cycle, violating the legal total order of S. Using basic two-party communication complexity, we show that legalizing the total order requires transferring $R \rightarrow w_1$ from S_1 to S_2. ■

We now extend e_1 with more read-only transactions, servers, and write requests, and apply the structure above to force more dependency metadata to transfer between servers. We then quantify this metadata and show that it violates C.

Proof of the NOCS Theorem. Suppose the system has $M^2 + 1$ servers $S_1, S_2, \ldots, S_{M^2+1}$. Let $R_1, R_2, \ldots, R_N$ be N read-only transactions that execute `ALG`, where each R_i sends a read request to S_1 and $M-1$ other servers, such that every server other than S_1 receives N/M read requests. (In practice $M^2 \ll N$, but our construction works for any $N, M \geq 1$.) The

specific mapping of read requests to servers is unimportant; we lay them out sequentially by transaction index below. Let $r_{i,j}$ be a read request of R_i assigned to S_j. We assign one read request from each of R_1 to $R_{N/M}$ to S_2, one read request from each of $R_{N/M+1}$ to $R_{2N/M}$ to S_3, and so on, restarting at R_1 after reaching R_N. Let $w_1, w_2, \ldots, w_{M^2+1}$ be M^2+1 simple writes issued to each server by a distinct client C_w that does not issue any read-only transactions. Suppose w_1 precedes all other writes, i.e., $w_1 \rightarrow w_j$ for $j = 2, \ldots, M^2+1$, and all read-only transactions are concurrent with all writes. Consider the execution e_*:

$$
\begin{array}{lll}
S_1: & r_{1,1}, \ldots, r_{N,1}, w_1 & \\
S_2: & & w_2, r_{1,2}, \ldots, r_{N/M,2} \\
S_3: & & w_3, r_{N/M+1,3}, \ldots, r_{2N/M,3} \\
\vdots & & \\
S_{M+1}: & & w_{M+1}, r_{N-N/M+1,M+1}, \ldots, r_{N,M+1} \\
S_{M+2}: & & w_{M+2}, r_{1,M+2}, \ldots, r_{N/M,M+1} \\
\vdots & & \\
S_{M^2+1}: & & w_{M^2+1}, r_{N-N/M+1,M^2+1}, \cdots, r_{N,M^2+1}
\end{array}
$$

By decomposing this execution into layers, we can inductively quantify the metadata required to process it. Let e_1 be the execution fragment containing all write requests and only the read requests of R_1. Let e_i contain the requests of e_{i-1} plus all read requests of R_i, for $i = 2, \ldots, N$. Thus $e_N = e_*$.

Lemma 3. *Processing e_k while satisfying N+O+S requires $\Omega(kM^2)$ metadata, for $k = 1, \ldots, N$.*

Proof Summary. The proof is by induction. For the base case of e_1, Lemma 2 requires us to transfer $R_1 \rightarrow w_1$ from S_1 to all $M-1$ servers targeted by R_1. We show that the write protocol cannot efficiently transfer this metadata, since it does not know which servers R_1 targets, and hence must send $R_1 \rightarrow w_1$ to all M^2 servers, or $\Omega(M^2)$ metadata. Alternatively, $r_{1,1}$ can convey the list of target servers, but due to asynchrony, a different execution could cause a different target server S_j to play the role of S_1, making it impossible to know which $r_{1,j}$ will appear before a write. Thus, every $r_{1,j}$ must include the list of M servers, requiring $\Omega(M * M) = \Omega(M^2)$ metadata. In the inductive step, we show that e_k cannot rely on previous metadata transferred in e_{k-1}, and thus requires an additional $\Omega(M^2)$ metadata. ■

Completion of the proof. By Lemma 3, $e_* = e_N$ requires $\Omega(NM^2)$ metadata. Since $R_1, \ldots, R_N$ issue NM read requests total, the amortized metadata required per read request is $\Omega(\frac{NM^2}{NM}) = \Omega(M)$, which is not constant, violating C. ■

4.2 The Broad Scope of NOCS

We prove NOCS is impossible in the specific setting of one-shot read-only transactions in failure-free systems. When it comes to an impossibility result, the more restricted the setting it is proved in, the stronger the result, because any setting that is more general is also subject to the impossibility result (the general setting *includes* the restricted setting as a special case). Thus, the NOCS Theorem also applies to more general settings, such as those with read-write transactions, multi-shot transactions, and/or failures.

4.3 NOCS Is Tight

While all properties are impossible to achieve together, we find that NOCS is "tight" in the sense that any combination of three properties is possible. Spanner's [10] read-only transactions are one-round, use constant metadata, but block reads in order to return strictly serializable results (O+C+S). Many systems use multiple non-blocking round trips to coordinate strongly consistent results (N+C+S), e.g., DrTM [49], RIFL [29]. To the best of our knowledge, no existing system provides strict serializability in one round of non-blocking communication (N+O+S). We present the design of such a system, PORT-SEQ, and a proof of its correctness in our technical report [35]. The design uses a centralized write sequencer to totally order writes, and requires a linear amount of metadata for read-only transactions. We are aware of two systems that have performance-optimal read-only transactions (N+O+C): MySQL Cluster [39] and the snapshot read API of Spanner. These systems provide weak consistency, however, as we discuss below.

5 NOCS Connects Theory with Practice

This section discusses the value of the NOCS Theorem in understanding the design space and in guiding system designs.

5.1 Theoretical Insights

Proving the impossible. NOCS is philosophically similar to other impossibility results like CAP and SNOW, in that it helps system designers avoid attempting the impossible and instead identifies a fundamental choice they must make: their system can either have performance-optimal read-only transactions or provide strict serializability, but not both.

Identifying the possible. The crux of NOCS's impossibility is that the real-time requirement of strict serializability forces read-only transactions to confront conflicting requests (Lemma 1). This suggests optimal performance could be possible with even *slightly* relaxed consistency models that do not require real-time ordering, and thus can avoid the unstable region. In particular, the second strongest consistency model we are aware of—process-ordered serializability [34]—does not require real-time ordering.

Yet, there is a large gap in the current design space. The only two existing systems whose read-only transactions are performance optimal provide weak consistency. MySQL Cluster's read-committed consistency does not isolate transactions. Spanner's snapshot read API can be used to get performance optimality, but it does not ensure clients see their own recent writes when used in this way (§10). Between these weak guarantees and strict serializability are

many stronger consistency models, such as read-atomic [5], causal consistency [31], and process-ordered serializability [34]. We bridge this gap by presenting the PORT design that provides performance-optimal read-only transactions and the strongest consistency to date: PORT provides process-ordered serializability in systems with only simple writes (§6), and it provides causal consistency in systems with write transactions (§8). (We conjecture causal consistency is the upper bound for performance-optimal read-only transactions when transactional writes are present.)

5.2 Guiding System Designs

NOCS is also useful in guiding system designs. First, to make a design performance-optimal, it must satisfy the NOC properties: each transaction must succeed using a single round of non-blocking messages with constant metadata. Therefore, the NOC properties indicate we must avoid validation-based and stabilization-based techniques to satisfy O, avoid techniques based on distributed lock management to satisfy N, and ensure the complexity of processing a read does not depend on the level of contention—i.e., the number of conflicting reads and/or writes—to satisfy C. Second, the NOCS Theorem suggests a path towards designing NOC protocols by avoiding how it derives its impossibility: read-only transactions should always execute on system states outside the unstable region. These implications of the NOC properties and the NOCS proof significantly reduced the design space of algorithm we needed to explore and led us to two high-level techniques for PORT: explicit ordering control and multi-versioning.

Explicit ordering control. There are two methods for ensuring reads avoid the unstable region by explicitly controlling the ordering of concurrent operations. First, reads can request versions of the data that lie before the unstable region begins, which orders a read-only transaction before ongoing writes. Second, servers can reorder operations when a read requests data in the unstable region.

Explicitly controlling ordering is not compatible with strict serializability because the real-time requirement forces a specific ordering of operations (Lemma 1) that cannot be communicated in a performance-optimal system (Lemma 3). Consistency models without the real-time requirement, however, might be compatible with an explicitly controlled ordering while satisfying NOC. PORT confirms this, by using versions clocks to capture this explicit ordering. PORT uses both types of explicit control on top of multi-versioning to provide its consistency guarantees and optimal performance.

Multi-versioning. Enabling reads to control what version of data they request requires multi-versioning on servers. Multi-versioning introduces storage overhead to temporarily keep additional version around, but this overhead is minor as storage is inexpensive and extra versions are not kept long. It also introduces some processing overhead to look up the correct version of data to return, reflected by our C property.

The need for multi-versioning to support efficient reads is not new. The existing performance-optimal systems, Spanner and MySQL Cluster, are multi-versioned. In fact, all existing systems whose read-only transactions are guaranteed to terminate—i.e., have a bounded number of retries and/or bounded blocking—are multi-versioned (Table 9). On the other hand, multi-versioning alone does not ensure optimal performance: most MVCC protocols require either extra on-path messages to query a timestamp oracle [6, 43], off-path messages to compute stable snapshots [3, 15], or blocking reads if the client-provided timestamp in MVTSO-based protocols points to the future [30, 45]. PORT's novelty is in how it uses version clocks to explicitly control ordering by manipulating the multi-versioning framework in order to achieve optimal performance.

6 PORT Design

PORT is a new system design that enables performance-optimal read-only transactions with process-ordered serializability, the strongest consistency to date.

Process-ordered serializability. Process-ordered serializability guarantees there exists a legal total order of transactions that respects the ordering of transactions within each process [34]. It is equivalent to sequential consistency [27] with the addition of transactional isolation. It preserves all the properties of strict serializability (§4) except for the real-time order across processes (clients). That is, it preserves the real-time order within each process, i.e., process order, and a total order across processes, but a client may not see the most recent updates of other clients. *Process order* ensures that each client interacts with the system monotonically, e.g., sees her own recent writes. *Total order* ensures that concurrent transactions are observed by all clients in the same order.

6.1 Version Clocks

This section describes *version clocks* (§6.1), a new specialized logical clock that tightly couples all the components of PORT (§6.2). Version clocks also allow us to avoid the work of applying some writes (*write omission*, §6.3) and limit the staleness of reads (*data freshness*, §6.4).

Version clocks are designed in the context of distributed storage systems and have two features: they ensure process order by concisely capturing the ordering constraints between requests and enable optimal performance by reading at the most recent snapshot in the stable region.

Enforcing process order. Version clocks take advantage of two observations. First, process order is a per-client order, and thus can be explicitly controlled by clients. Second, read and write requests have different semantics, i.e., writes modify system state while reads do not. Therefore, they should be treated differently: it is unnecessary to enforce an order among the read requests that observe the same system state.

Capturing the stable frontier. Version clocks follow the practical guidance of the NOCS Theorem (§5.2) to avoid the

```
1 Client Side
2 versionstamp = 0 # clock value
3 view[]           # max known versionstamp per server
4
5 # Sending requests
6 function get_vs_read():
7   versionstamp = tick(min{view[]}) # stable frontier
8   return versionstamp
9
10 function get_vs_write():
11   versionstamp++
12   return versionstamp
13
14 # Receiving a response msg from server svr
15 function recv_response(maxVS):
16   view[svr] = max{view[svr], maxVS}
17   if msg.for_write is true
18     versionstamp = tick(maxVS)
19   return
20
21 function tick(vs):
22   return max{vs, versionstamp}
23
24 Server Side
25 maxVS = 0 # max seeen versionstamp
26 # ... return maxVS when sending response msg
```

Figure 1: Pseudocode for version clocks.

unstable region by capturing the stable frontier. The stable frontier is the most recent snapshot in which all writes are in the stable region. Each server tracks the final versionstamp of its most recent write. A version clock tracks the minimum of such versionstamps across all servers the client has contacted, which is exactly the stable frontier the client knows. Version clocks direct read messages to the stable frontier when possible. PORT takes care of the cases when reads have to confront conflicting requests beyond the stable frontier. "Promotion" is used in systems with simple writes to advance the stable frontier beyond the versionstamp of an incoming read to ensure a total order. "Per-client ordering" is used in systems with write transactions to logically move a client's own writes before the stable frontier so the client can always safely read at the stable frontier (§8.2). Both techniques enforce the necessary order between concurrent reads and writes without blocking either reads or writes.

Clock structure. Figure 1 shows the pseudocode of version clocks. *versionstamp* stores the current clock value (line 2), which is embedded in every read/write message to explicitly control their ordering. When versionstamps are the same for two operations of the same type, the server orders them arbitrarily. When versionstamps for a read and a write are the same, the server orders the read after the write. A server responds with the highest versionstamp it has seen (line 26). A client uses *view* to track the highest versionstamps of the servers it has contacted (line 3) and uses them to find the stable frontier (line 7) before sending a read message (lines 6–8). *view* is updated upon receiving a response (line 16). If the response is for a write message, then the clock is advanced so that future read messages will have greater versionstamps than the write (lines 17–18), ensuring read-your-writes. Because versionstamps increase monotonically and reads have non-smaller versionstamps than earlier writes, version clocks preserve process ordering.

```
1 Client Side
2 function read_only_txn(<keys>):
3   vs = VersionClock.get_vs_read()
4   for k in keys # in parallel
5     vals[k], maxVS = read(k, vs)
6     VersionClock.recv_response(maxVS)
7   return vals   # replies to end user
8
9 function write(key, val):
10   vs = VersionClock.get_vs_write()
11   maxVS = write(key, val, vs)
12   VersionClock.recv_response(maxVS)
13   return        # replies to end user
14
15 Server Side
16 vers[keys][] # multi-versioned storage
17 function read(key, vs):
18   if vers[key][vs] exists
19     return vers[key][vs], VersionClock.maxVS
20   else # return nearest version to not block
21     near_vs = find_nearest_earlier(ver)
22     # ensure future writes have higher vs
23     vers[key].max_r_vs = max(vers[key].max_r_vs, vs)
24     return vers[key][near_vs], VersionClock.maxVS
25
26 function write(key, val, vs):
27   if vs <= vers[key].max_w_vs
28     return VersionClock.maxVS # omit write
29   if vers[key].max_r_vs >= vs
30     vs = max_r_vs + 1 # commit after promoted versions
31   vers[key][vs] = val
32   vers[key].max_w_vs = vs
33   if vs > VersionClock.maxVS
34     VersionClock.maxVS = vs
35   return VersionClock.maxVS
```

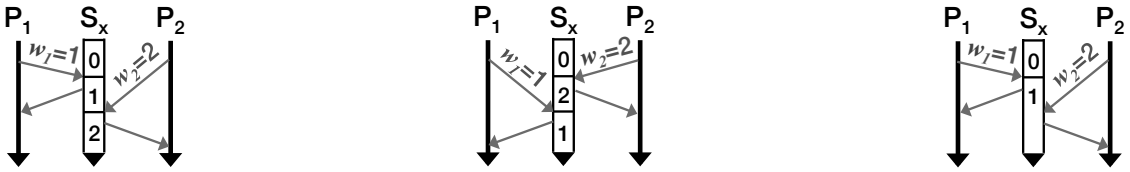
Figure 2: Pseudocode for PORT.

6.2 Basic PORT Design

The basic PORT design includes a multi-versioning framework, a read-only transaction algorithm, and a write algorithm. We co-design these components tightly by leveraging version clocks. Figure 2 shows PORT's pseudocode.

Client library. The read-only transaction and write algorithms are executed by a client library. For each read-only transaction or write, the client obtains a versionstamp from its version clock and embeds it in the request message(s). This per-client versionstamp decides which system version on the servers the operation must read (or write) to ensure the client's process order (lines 3, 10). The server-side logic ensures a total order on top of the process order on each client to guarantee process-ordered serializability.

Multi-versioning framework. Servers store written values in a multi-versioning framework (line 16). Since PORT uses version clocks to track the ordering between operations, it is natural and efficient to index the historical values of each data item with versionstamps. In this way, the multi-versioning framework and transaction layer are nicely coupled via versionstamps. We omit a detailed discussion of garbage collection, which uses standard mechanisms similar to those used to provide at-most-once semantics.



(a) orders w_1 before w_2 by arrival. **(b) orders w_2 before w_1 by arrival.** **(c) orders w_2 before w_1 by omission.**

Figure 3: Space-time diagrams showing three executions of writes w_1 and w_2 that are concurrent and conflicting. The value underneath S_X indicates the value stored by the server. Process-ordered serializability allows w_1, w_2 to be ordered either way. This enables us to omit w_2 in (c) because it is equivalent to the ordering in (b), i.e., (w_2, w_1).

Read-only transactions. To process a read request, a server executes it against the system version specified by its versionstamp. Executing a read is thus equivalent to returning the value indexed at versionstamp. If the server has the requested version, then the read is inside the stable region and it returns the version directly (lines 18, 19). Otherwise, it uses promotion to ensure a total order between the read and any concurrent writes at the specified versionstamp, without blocking either the read or write (lines 20–24).

Promotion logically copies the value of the nearest earlier version to all empty positions between that version and the one requested by versionstamp. Logical versions are used as placeholders to ensure a total order: once a version has been read by any client, no earlier versions can be modified to ensure different clients observe them in the same order. For example, if a read request has $vs = 4$ and the data item has committed values at $vs = 1, 2$, the version at $vs = 2$ is the nearest earlier version and is promoted to positions $3, 4$. A conflicting write at $vs = 3, 4$ will be "bumped up" to $vs = 5$ when it arrives. We implement promotion with a single variable (line 23) that marks earlier positions as immutable.

Writes. When receiving a write request, a server finds the position specified by the write's versionstamp in the multi-versioning framework. If the position is empty, then the write is applied at the versionstamp (line 31). If the position has been marked immutable by read promotion, the server finds the next available position to write the version at (lines 29–31). The write protocol also includes a mechanism for safely skipping concurrent writes (lines 27–28), discussed next.

6.3 Write Omission

Write omission is a special conflict resolution mechanism that skips an incoming write if it is concurrent with an already applied write. Omitting a write is desirable because it saves the computation needed to apply it, reduces the number of stored versions, and saves the work of replicating it.

Write omission is safe. Consistency models in general, and process-ordered serializability specifically, allow conflicting writes to be ordered either way. For instance, if two processes concurrently issue w_1 : `write(x = 1)` and w_2 : `write(x = 2)`, then they can be ordered as either (w_1, w_2) or (w_2, w_1). Typically, systems apply writes in the order that they arrive, e.g., w_1 then w_2. But if instead we use the opposite order, then this is equivalent to omitting w_2, as shown in Figure 3: skipping the later write is equivalent to ordering it before the earlier write and immediately overwriting it with the latter. Write omission does not affect the total order requirement: all clients observe concurrent writes in the same order, because omitted writes are never seen by any client.

Knowing a write is concurrent. Version clocks enable PORT to identify when writes are concurrent, allowing a later concurrent write to be omitted. PORT omits an incoming write if its versionstamp, vs_{omit}, is less than or equal to the highest committed versionstamp of the data item, $vs_{highest}$ (lines 27–29). The write with the highest committed versionstamp cannot have happened-before [26] the omitted write because $vs_{highest} \geq vs_{omit}$. More specifically, version clocks guarantee the invariant: if write x happens-before write y, then $vs_x < vs_y$. The omitted write cannot have happened-before the write with the highest committed versionstamp because it has not happened yet. Therefore, the two writes are concurrent, and it is safe to omit the incoming write.

Omitting a write is equivalent to applying it immediately before the write with the highest versionstamp. A client's future reads must observe the "higher" write if its own write was overwritten in this way. Therefore, the server returns the versionstamp of its highest applied write to the client (line 29), which uses it to update its versionstamp as normal.

6.4 Keeping Reads Fresh

To avoid the unstable region, we must sometimes return values staler than what strict serializability would return (§5.2). PORT limits data staleness in two ways, neither of which incurs extra messages, blocking, or non-constant metadata. That is, they do not forfeit optimal performance (NOC).

Reducing staleness with version clocks. Instead of naively returning versions far behind the stable frontier, version clocks try to track the stable frontier precisely. They use *view* to track the most recent versionstamp on each server a client has contacted, so a client's version clock never ticks slower than the servers it is aware of. This significantly improves the freshness of data requested by read-only transactions.

Reducing staleness via co-location. Many storage systems co-locate "end users" on the same client machine [12, 18,

40], i.e., each client (machine) has many sessions (threads), one per end user. We leverage co-location to help user sessions keep each other fresh by sharing one version clock among them on the same client, which ensures no user session is staler than the freshest session it is co-located with.

6.5 Correctness and Generality

The only technique PORT relies on is version clocks, which can easily be added to systems with existing physical/logical clocks, or implemented from scratch. We demonstrate both by applying PORT to a system without transactions (shown by Scylla-PORT) and a system with existing sub-optimal read-only transactions (shown by Eiger-PORT). We present a proof of correctness for PORT in our technical report [35].

Failures. PORT can tolerate server failures using typical techniques such as state machine replication [46]. To tolerate client—i.e., frontend—failures, clients can send versionstamps back to end-user machines that then include the versionstamp in subsequent requests to the application (e.g., via cookies). This ensures process ordering is maintained even if an end user's later requests go to a different frontend due to load-balancing or frontend failure.

7 PORT Implementation and Evaluation

This section discusses Scylla-PORT, the implementation of PORT on a clean slate base system.

7.1 Implementation

We build PORT on ScyllaDB [47], a clean slate, non-transactional base system that supports only simple reads and simple writes. ScyllaDB is a production system that serves as a drop-in replacement for Cassandra [25] and provides an order-of-magnitude better performance. It is well-engineered and aggressively-optimized for performance, including a new implementation in C++14, core-level sharding that avoids cross-core locking and context switches, and customized lock-free data structures.

Rationale and takeaways. We chose to implement PORT on ScyllaDB for three reasons. First, it stresses the efficiency of PORT: as a highly efficient baseline system, it is sensitive to any additional overheads, and thus amplifies any performance cost introduced by PORT. Second, ScyllaDB is single-versioned. The negligible performance overhead shown in our evaluation includes the cost of making it multi-versioned (§5.2), which shows the efficiency of co-designing the multi-versioning framework and the transaction layer enabled by version clocks. Third, PORT is compatible with all the customized engineering decisions of ScyllaDB, which demonstrates the generality of the design of PORT.

7.2 Evaluation Overview

We evaluate Scylla-PORT against ScyllaDB (the clean slate, non-transactional base system) and Scylla-OCC (an implementation of OCC atop ScyllaDB). We compare their throughput, latency, scalability, and quantify data staleness.

Scylla-OCC. We implemented a variant of OCC optimized for read-only transactions, similar to Rococo's read-only transaction algorithm [37]. It includes an initial round of optimistic reads and then a validation round. If the values read in the optimistic round match the values in the validation round the transaction succeeds. If not, the read-only transaction is aborted and retried. This variant has strictly better performance than traditional distributed OCC because it avoids the need for distributed commit: its best case is two rounds compared to traditional distributed OCC's best case of three rounds (read, validate/prepare, commit).

Code. We implemented our server-side logic in ScyllaDB's codebase (release 2.1-RC3) in C++14 and our client-side logic in the Java Thrift client of the YCSB benchmark (release 0.10.0) [9]. Version clocks are implemented on both servers and clients. Scylla-PORT adds ~1,300 LOC.

Experimental setting. We run experiments on Emulab [50]. Each machine has two 2.4 GHz 8-Core Xeon CPUs, 64 GB RAM, and a 10 Gbps network interface. We use a single datacenter setting. All experiments, except for scalability tests, use 8 servers loaded by 8 client machines. The scalability tests use up to 64 machines. Each client issues 10 million requests in each experiment, which takes 5–10 minutes to complete, sufficiently long to minimize warm-up and cool-down effects and provide stable results. Experiments are CPU-bound on servers.

Configuration and workloads. We use YCSB's standard workloads B (read-heavy, 95% reads) and C (read-only) with customized read-to-write ratios of up to 25% writes. We use YCSB's default parameters: 1 million records, 10 fields per record, 100 B values per field, and Zipf constant of 0.99. Each request (a read-only transaction or a group of simple writes) accesses 5 records and all fields in each record.

Results summary. Transactional overhead is generally evident with read-write conflicts and under skewed workloads, so we focus our evaluation in such scenarios to amplify Scylla-PORT's cost. Our results show that Scylla-PORT can almost match its performance to that of non-transactional ScyllaDB: 1–3% overhead in throughput and latency in most settings and less than 8% even in the worst case. Scylla-PORT outperforms OCC by an order-of-magnitude in such contended scenarios due to OCC's retries, and outperforms OCC under low contention (OCC's best case) by at least two times. Scylla-PORT scales as well as ScyllaDB and scales better under contention. More than 40% of its reads return fresh values.

7.3 Throughput and Latency

Figure 4a shows the overall performance of the systems as we gradually increase the system load by using more closed-loop client threads. Scylla-PORT has similar performance to the baseline ScyllaDB. Their largest difference before Scyl-

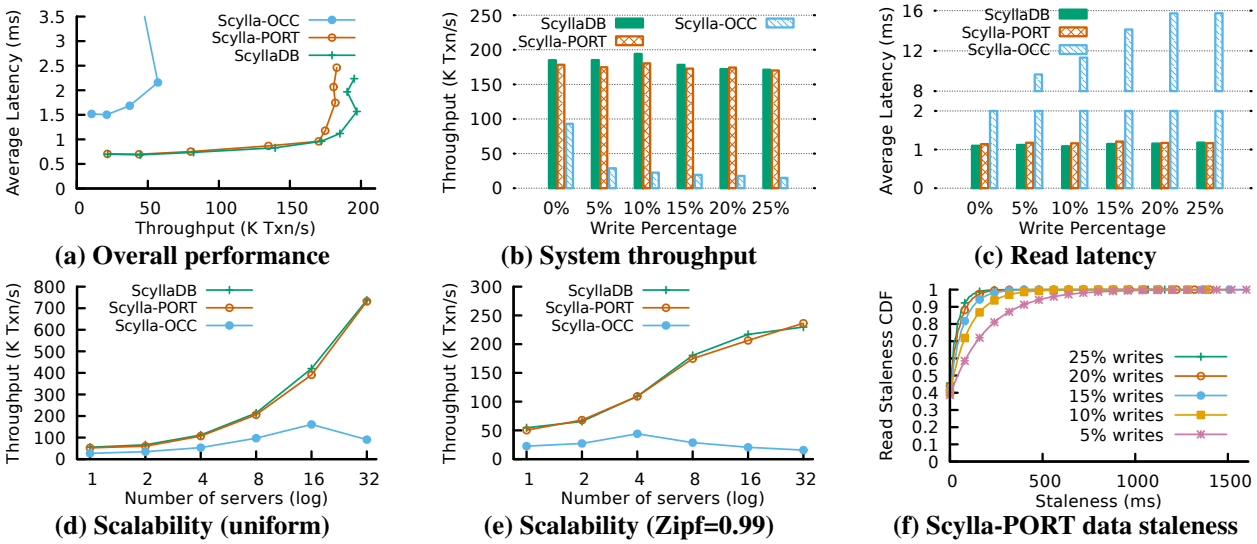


Figure 4: The performance of Scylla-PORT closely matches non-transactional ScyllaDB and is significantly better than OCC, Scylla-PORT scales even better than ScyllaDB with skewed workloads, and half of its reads return fresh data.

laDB becomes overloaded is evident with 32 client threads: 5.6% in throughput and 5% in latency. All later experiments report throughput and latency at this operating point, i.e., with 32 client threads. OCC initially has latency that is twice that of ScyllaDB and Scylla-PORT because it takes at least two rounds to complete instead of one. As load increases, OCC's latency increases quickly and its throughput decreases slightly because contention forces it to retry.

Varying write percentage. Figure 4b and 4c show the throughput and latency as we vary the read-to-write ratio. Scylla-PORT's throughput is within 4% of ScyllaDB's for five of the experiments and within 7% for the remaining one. Similarly, its latency is within 2% (20 µs) of ScyllaDB's for two of the experiments and within 7% (107 µs) for the other four. As the write percentage increases, the overhead disappears because of write omission: doing slightly more work during reads is offset by doing less work during writes. When there are only reads, Scylla-PORT has double the throughput and half the latency of OCC because OCC's read-only transactions require at least two rounds. With writes, OCC's performance drops quickly due to retries.

7.4 Scalability

Figure 4d compares the scalability of the three systems under a uniform workload as we increase the number of servers while increasing the number of clients to keep the servers CPU-bound. Scylla-PORT scales as well as ScyllaDB; the differences in throughput are negligible. Interestingly, Scylla-PORT outperforms ScyllaDB under a skewed workload, as shown in Figure 4e. ScyllaDB stops scaling at 16 servers because the server holding the hottest keys becomes the bottleneck, and adding more servers does not help. (We have confirmed this finding with ScyllaDB's developers.) Scylla-PORT scales better than ScyllaDB under skewed workloads because it can avoid the work of some writes to the hottest keys due to write omission. Since write omission only applies to conflicting writes, this rarely occurs under a uniform workload. OCC initially shows a similar scaling pattern starting from its lower throughput. OCC's scaling stops, however, as more concurrent clients accessing the same keys lead to higher contention and thus more retries.

7.5 Data Staleness

Figure 4f shows the staleness of Scylla-PORT under a skewed workload with varying write percentages. Staleness is measured relative to strict serializability, which always has a staleness of 0: it is the amount of time since a newer version has been committed. For example, if v_0, v_1 are consecutive versions, v_0 is returned at 0:05, and v_1 committed at 0:00, then the staleness of v_0 is 5 seconds.

Scylla-PORT returns the most recent data ~40% of the time, and 90% of reads return values no staler than 500 ms. Scylla-PORT returns fresher data as the write percentage increases because version clocks advance versionstamps more frequently when there are more writes. Scylla-PORT leverages version clocks to precisely capture the stable frontier, but does not utilize client co-location. Sharing one clock among co-located user sessions would further decrease staleness, but also decreases the rate at which write omission can be used. We leave investigating this tradeoff to future work.

7.6 Low Contention Evaluation

We focused here on high contention workloads because those are where any differences between Scylla-PORT and ScyllaDB would appear. Scylla-OCC did poorly in this setting as is expected because OCC is better suited to low contention

settings. We present the results of evaluating the three systems under low contention in our accompanying technical report [35]. Even in that setting, Scylla-PORT significantly outperforms Scylla-OCC with at least double the throughput and at most half the latency because Scylla-PORT always finishes in one round while OCC's best case is two rounds.

8 Improving an Existing System

This section adapts PORT to improve Eiger, an existing system that has both read-only and write transactions.

8.1 Eiger Overview and Rationale

Eiger is a geo-replicated, causally consistent system that has read-only transactions and write transactions. Each machine implements a Lamport clock and attaches a Lamport timestamp to each committed write that is guaranteed to be larger than any earlier write it causally depends on. Eiger's write transaction protocol is a variant of two-phase commit [21, 28] that always commits. Eiger's read-only transaction protocol takes between one and three non-blocking rounds of communication. If there are no concurrent write transactions, it completes in a single round. Otherwise, it requires a second round of messages to a subset of the servers, followed by a third round if the concurrent write transactions are still pending when the second-round requests arrive. In the third round, each read request needs to query the states of all write transactions it conflicts with, and thus the required metadata increases linearly with respect to the number of conflicting write transactions.

Rationale. We choose Eiger as a base system because of its guarantees and the efficiency of its read-only transactions. First, it provides causal consistency, not strict serializability, so it may be possible to add performance-optimal read-only transactions to it. Second, it includes write transactions, which present a new challenge for the PORT design. Third, it is the only system with write transactions and causal (or stronger) consistency that completes read-only transactions in a bounded number of non-blocking rounds of communication (Figure 9). Finally, its read-only transactions often complete in a single non-blocking round, making them a more difficult baseline than other algorithms such as OCC.

8.2 Eiger-PORT

Eiger's read-only transactions are non-blocking, require up to three rounds of on-path communication, and use linear-sized metadata in the third round. We make them performance-optimal by making them always finish in one round using only constant metadata. The major challenge is to ensure write isolation, i.e., return a system state that is either before all updates in a write transaction or after.

More specifically, when a read-only transaction must read beyond the stable frontier, e.g., to ensure read-your-writes, PORT reorders the read-only transaction and the conflicting writes without blocking by using "promotion" (§6.2). However, promotion does not work for Eiger because it cannot ensure that all writes in the same write transaction are promoted at the same time since they can be on different servers. Our solution, *per-client ordering*, enables clients to observe conflicting writes in different orders, as allowed by causal consistency. Specifically, it pulls back any of a client's recent writes that are beyond the stable frontier. This allows the client to read at the stable frontier while also always seeing their own writes. Figures 5, 6, and 7 show the pseudocode, written in a way that favors clarity over efficiency.

```
1 Client Side
2 lst_map[][]   # maps server to its local safe time
3 gst           # global safe time
4
5 function read_only_txn(<keys>):
6   gst = get_read_ts(min{lst_map.valueSet()})
7   for k in keys  # messages in parallel
8     vals[k], lst = read(k, gst, cl_id)
9     lst_map[k.server] = lst # lst is monotonic
10  return vals
11
12 function write_txn(<keys, vals>):
13  for k, v in <keys, vals>  # in parallel
14    if k.server is coord # the coordinator
15      lst = write_coord(k, v, cl_id, gst)
16    else # a cohort
17      lst = write_cohort(k, v, cl_id, gst)
18    lst_map[k.server] = lst # lst is monotonic
19  return
20
21 function get_read_ts(ts):
22  return max{ts, gst}
```

Figure 5: Client-side pseudocode for Eiger-PORT.

```
1 Server Side (Read-Only Txn)
2 lst             # local safe time, updated upon writes
3
4 function read(k, rts, cl_id):
5   ver = DS[k].at(rts) # vers are sorted by commit_t
6   for v in DS[k].newer_than(ver.commit_t)
7     # ensure read-your-writes, from newer ver to old
8     if v.cl_id == cl_id
9       return v.val, lst
10  if ver.cl_id != cl_id
11    return ver.val, lst
12  else # ensure write isolation
13    v = find_isolated(ver)
14    return v.val, lst
15
16 function find_isolated(ver):
17  # iterate from newer version to old
18  while v in DS[k].newer_than(ver.gst)
19    and v in DS[k].older_than(ver.commit_t)
20    if v.cl_id != ver.cl_id
21      return v
22    else
23      return find_isolated(v)
24  return ver
```

Figure 6: Read-only transaction logic for Eiger-PORT.

Client-side logic. Figure 5 shows the client-side logic. Each client maintains two variables (lines 2, 3). *lst_map* tracks the local safe time, *lst*, of each server. Global safe time, *gst*, is the minimum *lst* across all servers (line 6) and advances monotonically. *gst* is used as the read timestamp for each

```
1 Server Side (Write Txn)
2 lst              # local safe time
3 pending_wtxns    # uncommitted write txns
4 DS[][]  # multi-versioned k-v data store
5
6 function write_coord(k, v, cl_id, gst):  # coordinator
7   # PREPARE
8   ver, prepared_t = prepare_write(k, v, cl_id, gst)
9   # ... get yes-vote-msgs from all cohorts
10  # COMMIT
11  commit_t = max{yes-vote-msgs.prepared_t, prepared_t}
12  commit-msg = {"commit", commit_t}
13  # ... send commit-msg to all cohorts
14  commit_write(ver, commit_t)
15  return lst
16
17 function write_cohort(k, v, cl_id, gst):  # cohort
18  # PREPARE
19  ver, prepared_t = prepare_write(k, v, cl_id, gst)
20  yes-vote-msg = {"yes", prepared_t}
21  # ... send yes-vote-msg to coordinator
22  # ... wait for commit-msg
23  # COMMIT
24  commit_t = commit-msg.commit_t
25  commit_write(ver, commit_t)
26  return lst
27
28 function prepare_write(k, v, cl_id, gst):
29  pending_t = LamportClock.current()
30  pending_wtxns.append(pending_t)
31  LamportClock.advance()
32  ver = DS[k].create_new_ver(v, cl_id, gst, pending_t)
33  ver.is_pending = true
34  return ver, LamportClock.current()
35
36 function commit_write(ver, commit_t):
37  ver.commit_t = commit_t
38  ver.is_pending = false
39  pending_wtxns.remove(ver.pending_t)
40  if pending_wtxns is empty
41    lst = LamportClock.current()
42  else
43    lst = pending_wtxns.head() # min of pending_wtxns
44  return
```

Figure 7: Write transaction logic for Eiger-PORT.

read-only transaction. Both *lst* and *gst* are Lamport timestamps as used in Eiger. A client sends all read requests in a read-only transaction in parallel. Each read request includes the key, the read timestamp *gst*, and the unique identifier of this client (line 8). The server responds with the requested value and *lst* on that server. A client issues a write transaction by sending the write requests in parallel (lines 12–19). One server is randomly chosen as the coordinator (line 14) for 2PC with the others as cohorts. Each write request contains the key, the value, the client ID, and the client's current *gst* (lines 15, 17). *gst* specifies the stable frontier this write transaction causally depends on. The client updates *lst_map* after each read/write request (lines 9, 18).

Write transactions. Figure 7 shows the server-side logic of write transactions. When a server receives a write request, it records the current Lamport time (line 29) and creates a new pending version (lines 8, 19, 32, 33). *pending_wtxns* tracks ongoing write transactions by keeping an ordered list of *pending_time*s. The running minimum of *pending_wtxns* is the *lst* on this server, i.e., no pending writes exist before *lst*. Because Lamport clocks advance monotonically, insertion, removal, and fetching the minimum of *pending_wtxns* have a cost of O(1). At the end of the "prepare" phase of 2PC, each cohort sends a yes-vote message to the coordinator, which includes the *prepared_time* of this pending write transaction. *prepared_time* is guaranteed to be greater than *pending_time* by clock ticking (line 31).

To commit a write transaction, the coordinator calculates the commit time by taking the maximum across all *prepared_time*s (line 11) and then sends a commit message to the cohorts and commits its local pending version (lines 13, 14). When a cohort receives the commit message, it commits its local pending version (lines 25, 38) with the commit time (lines 24, 37). It then removes this write transaction's *pending_time* from *pending_wtxns* and updates *lst* (lines 39–43). The server returns its *lst* to the client upon commit. Eiger-PORT made minimum changes to Eiger's write transactions, i.e., the management of *pending_wtxns*.

Read-only transactions. Figure 6 shows the server-side logic of read-only transactions. When a server receives a read request, it finds the version at the read timestamp, *rts* (line 5), and checks if the same client has made a recent write later than *rts*. It returns the most recent write by the same client to ensure read-your-writes (lines 6–9). If the version at *rts* was written by the same client, then we need to ensure write isolation by checking whether there exist any versions between the version's *gst*, which is the snapshot time the version depends on, and the version's *commit_t* (lines 18, 19). If there exists such a version written by a different client, then that version is returned to satisfy write isolation (lines 20, 21). We need to do this recursively, but our implementation uses a loop instead for better performance. To ensure write isolation (lines 16–24), we go through the multi-versioned data store once, which has the same cost as finding a particular version by timestamp in other algorithms, e.g., MVCC.

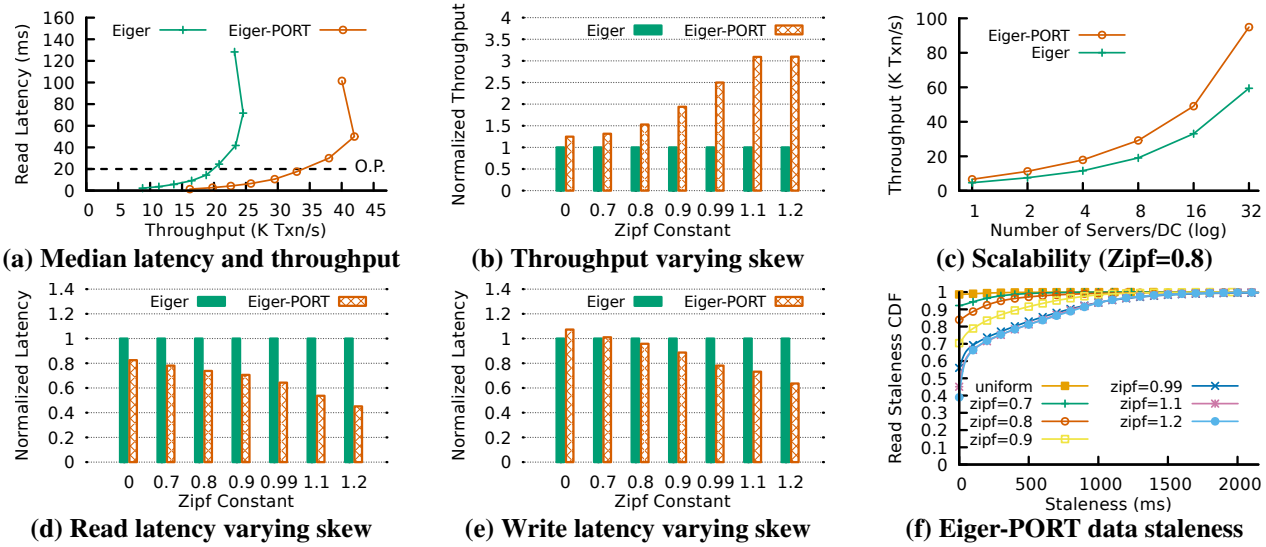
Correctness. We show the correctness of Eiger-PORT by proving that any execution in Eiger-PORT satisfies the causal ("happened before") relation [26] and write isolation for write transactions. We present the full proof in the technical report [35].

9 Eiger-PORT Evaluation

We evaluate Eiger-PORT against Eiger, showing its throughput and latency improvement as well as its data staleness.

Implementation. We implemented Eiger-PORT as a modification to Eiger's code base, which is built on top of Cassandra [25] and written in Java. Eiger-PORT adds ~1000 LOC.

Experimental setting. We try to match Eiger's original experimental setup. We run all experiments on Emulab [50], similar to the now-decommissioned PRObE testbed [19] Eiger used. Each machine has one 2.4 GHz Quad-Core Xeon CPU, 12 GB RAM, and a 1 Gbps network interface. We run 5 trials for each data point, each lasting 65 seconds, and report the median. We exclude the first and last 15 seconds to



(a) Median latency and throughput (b) Throughput varying skew (c) Scalability (Zipf=0.8)

(d) Read latency varying skew (e) Write latency varying skew (f) Eiger-PORT data staleness

Figure 8: Throughput, latency, scalability, and staleness of Eiger-PORT: up to 3× throughput improvement and 60% latency reduction compared to Eiger, better scalability, and low data staleness. All latencies are median latencies.

avoid artifacts due to warm-up, cool-down, and imperfectly synchronized clients. All experiments are CPU-bound.

Configuration and workloads. We use the same setting as Eiger: two logical datacenters co-located in the testbed. Each datacenter has eight server machines, and uses eight client machines to load the servers. The second datacenter is used as a replica, which applies updates replicated from the first datacenter. We use the dynamic workload generator from Eiger with the same default values: 1 million keys, 128-byte values, 5 columns per key, 5 keys per operation, and a write percentage of 10% unless otherwise specified. We also use a Zipf traffic generator with a default value of 0.8.

9.1 Performance Improvement

Results summary. Eiger-PORT significantly improves the performance of Eiger under different workloads, without degrading write performance: 2× and 3× throughput improvement under mild and high skew, respectively, and 20%–60% latency reduction. The performance improvement comes from Eiger-PORT's fewer on-path messages and less metadata to process. The improvement is larger in contended workloads because Eiger is more likely to require more than one round and more metadata in the third round when there are more conflicting write transactions.

Throughput improvement. Figure 8a shows the median read latency and system throughput as we double the number of closed-loop client threads loading the system (from 2 to 512). It shows that Eiger-PORT performs strictly better than Eiger: it achieves higher throughput with the same latency and lower latency with the same throughput. We run all other experiments in Figure 8 with 32 threads, representing an operating point with reasonably low latency (< 20 ms), i.e., at line "O.P." in Figure 8a. The improvements are more profound at higher loads. Figure 8b shows normalized throughput with different skew; the improvement stops increasing after Zipf value 1.1, where a single server becomes the bottleneck. Figure 8c shows Eiger-PORT scales better than Eiger due to fewer messages in the system.

Latency improvement. Figure 8d shows the normalized median read latency as we vary skew. Eiger-PORT achieves 20% lower latency under uniform workloads and up to 60% lower latency under contended workloads. Figure 8e shows that Eiger-PORT achieves lower write latency even though we did not intentionally improve writes. The lower latency comes from less queuing delay for writes because reads are faster and there are fewer messages in the system. This demonstrates that PORT can make read-only transactions performance-optimal without making writes more costly.

9.2 Data Staleness

Figure 8f quantifies the read staleness in Eiger-PORT. Staleness is measured relative to strict serializability as in Scylla-PORT's evaluation. Even with high skew, over 40% of Eiger-PORT's read-only transactions return up-to-date values, and over 90% of reads experience less than 1 s staleness. Eiger-PORT tends to return staler data than Scylla-PORT because the stable frontier moves more slowly in Eiger/Eiger-PORT: write transactions take longer to commit than simple writes.

10 Related Work

This section examines existing read-only transactions with the NOCS Theorem, reviews impossibility results, and discusses the move from latency to performance optimality.

Bridging the gap in the design space. We use the NOCS Theorem as a lens to better understand existing systems and show a set of representative systems in Figure 9. We find

System	N	O	C	S	W
Performance-optimal					
Scylla-PORT *	✓	✓	✓	POS	×
Eiger-PORT *	✓	✓	✓	Causal	✓
Spanner-Snap [10]*	✓	✓	✓	SR	✓
MySQL Cluster [39]*	✓	✓	✓	RC	✓
One fewer performance property for stronger guarantees					
Spanner-RO [10]*	×	✓	✓	✓	✓
DrTM [49]*	✓	≥ 1	✓	✓	✓
RIFL [29]	✓	≥ 2	✓	✓	✓
Sinfonia [1]	✓	≥ 2	✓	✓	✓
Candidates for improvement in performance and/or guarantees					
TAPIR [51]*	×	✓	✓	Ser	✓
Pileus-Strong [48]	×	2	✓	✓	✓
Rococo-SNOW [34]*	×	✓	Linear	✓	✓
COPS-SNOW [34]*	✓	Off-path	Linear	Causal	×
COPS [31]*	✓	≤ 2	Linear	Causal	×
RAMP-F|H [5]*	✓	≤ 2	Linear	RA	✓
RAMP-S [5]*	✓	2	✓	RA	✓
Eiger [32]*	✓	≤ 3	Linear	Causal	✓
Janus [38]	×	≤ 2	Linear	✓	✓
Callinicos [41]	×	2	Linear	✓	✓
Occult [36]	✓	≥ 1	✓	PC-PSI	✓
Rococo [37]*	×	≥ 2	✓	✓	✓
Contrarian [13]*	✓	2	✓	Causal	×
GentleRain [15]*	×	≤ 2 + off-path	✓	Causal	×
Cure [3]	×	Off-path	✓	Causal	✓
MVTSO [30, 45]	×	✓	✓	Ser	✓

Figure 9: A review of existing systems through the lens of NOCS. Asterisks denote specialized read-only transaction algorithms. *W* denotes write transactions.

a large gap in the design space. The only existing systems that have performance-optimal read-only transactions provide weak consistency (§4.3). MySQL Cluster [39] provides read-committed, which does not isolate transactions. Spanner's snapshot reads API [10] cannot always guarantee non-blocking read-your-writes. Suppose a client updates key k in a read-write transaction with commit timestamp ts, and then immediately performs a read-only transaction involving a set S of keys that includes k. To ensure read-your-writes, the client must use a timestamp greater than or equal to ts for its read-only transaction. But doing so may block since other keys in S may be involved in a read-write transaction that is in the midst of two-phase-commit with a commit timestamp less than ts. That is, Spanner must use its externally consistent read-only transaction API, which may block reads in such cases to ensure read-your-writes.

We bridge this gap in the design space with PORT, the first design that provides performance-optimal read-only transactions and the strongest consistency to date.

Other read-only transactions. Some systems choose to trade one performance property for stronger guarantees [1, 10, 29, 49] but still reside on the "tight boundary" of the NOCS Theorem. Many systems neither are performance-optimal nor provide the strongest possible guarantees [3, 5, 13, 15, 31, 32, 34, 36], and thus could potentially be improved by our PORT design.

Impossibility results. Our NOCS Theorem is philosophically similar to other impossibility results, e.g., FLP [17], CAP [7, 20], and SNOW [34], in that it saves system designers' effort from trying the impossible. The most relevant result is the SNOW Theorem, which we discuss next.

The move from latency to performance. SNOW [34] showed tradeoffs in the design space of read-only transactions with a focus only on latency. It proved optimal latency is impossible if the system is strictly serializable and has write transactions. This work aims for a more complete understanding of the tradeoffs in the design of read-only transactions by considering latency and throughput. The move from latency to performance has two takeaways.

First, optimal latency neither translates to nor forfeits optimal throughput. The former is shown by the two systems built with SNOW, which provided lower latency at the cost of lowering throughput. The latter is shown by our new designs that achieve both optimal latency and optimal throughput. What really matters is a complete understanding of the tradeoff between performance and consistency and its insights for designs—the major contributions of this work.

Second, higher demand for performance, e.g., the move from latency only to both latency and throughput, suggests higher difficulty in providing stronger guarantees. Optimal latency is possible in strictly serializable systems without write transactions, but optimal performance is not.

11 Conclusion

Distributed storage systems are a fundamental building block of large-scale web services. They rely on read-only transactions to provide consistent views of sharded data. Our NOCS Theorem proves that read-only transactions cannot have optimal performance in strictly serializable systems. We presented PORT, a performance-optimal read-only transaction design that provides the strongest consistency to date. We applied PORT to design Scylla-PORT and Eiger-PORT. Scylla-PORT has minimal performance overhead compared to its non-transactional baseline. Eiger-PORT significantly improves the performance of its transactional base system.

Acknowledgments

We would like to thank our shepherd, Jinyang Li, for her invaluable feedback that improved this work. We thank the anonymous reviewers for their careful reading of our paper and their many insightful comments and suggestions. We are also grateful to Christopher Hodsdon, Theano Stavrinos, and Jeffrey Helt for their feedback on earlier stages of this work. Our evaluation at scale was made possible by the Emulab testbed. This work was supported by NSF award CNS-1824130 as well as a gift from Microsoft Research.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2007.

[2] M. K. Aguilera, J. B. Leners, and M. Walfish. Yesquel: scalable SQL storage for Web applications. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, Jun 2016.

[4] S. Almeida, J. Leitao, and L. Rodrigues. ChainReaction: a causal+ consistent datastore based on chain replication. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, Apr 2013.

[5] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *ACM Special Interest Group on Management of Data (SIGMOD)*, Jun 2014.

[6] C. Binnig, S. Hildenbrand, F. Färber, D. Kossmann, J. Lee, and N. May. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *The VLDB journal*, 23(6):987–1011, 2014.

[7] E. A. Brewer. Towards robust distributed systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, Jul 2000.

[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, Jun 2013.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, Jun 2010.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. F. andSanjay Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2012.

[11] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (ATC)*, Jun 2012.

[12] Developer Blog. Twemproxy: A fast, light-weight proxy for memcached. `https://blog.twitter.com/developer/en_us/a/2012/twemproxy.html`, 2012.

[13] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel. Causal consistency and latency optimality: friend or foe? In *International Conference on Very Large Data Bases (VLDB)*, Aug 2018.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *ACM Symposium on Cloud Computing (SoCC)*, Oct 2013.

[15] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *ACM Symposium on Cloud Computing (SoCC)*, Nov 2014.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[18] H. Fugal, A. Likhtarov, R. Nishtala, R. McElroy, A. Grynenko, and V. Venkataramani. Introducing mcrouter: A memcached protocol router for scaling memcached deployments. `https://engineering.fb.com/core-data/introducing-mcrouter-a-memcached-protocol-router-for-scaling-memcached-deployments/`, 2014.

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

[20] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[21] J. N. Gray. Notes on database systems. IBM Research Report RJ2188 (Feb.1978), 1978.

[22] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[23] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. In *Proceedings of the VLDB Endowment (PVLDB)*, Aug 2008.

[24] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[25] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[27] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 1979.

[28] B. Lampson and H. Sturgis. Crash recovery in a distributed storage system. Xerox Palo Alto Research Center, 1979.

[29] C. Lee, S. J. Park, A. Kejriwal, S. Matsushitay, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[30] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR)*, Jan 2015.

[31] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2011.

[32] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2013.

[33] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[34] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[35] H. Lu, S. Sen, and W. Lloyd. Performance-optimal read-only transactions (extended version). Technical Report TR-005-20, Princeton University, Computer Science Department, 2020.

[36] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar 2017.

[37] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2014.

[38] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[39] MySQL. MySQL :: MySQL Cluster CGE. `https://www.mysql.com/products/cluster/`, 2016.

[40] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2013.

[41] R. Padilha, E. Fynn, R. Soulé, and F. Pedone. Callinicos: Robust transactional storage for distributed data structures. In *USENIX Annual Technical Conference (ATC)*, Jun 2016.

[42] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.

[43] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2010.

[44] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: core-aware thread management. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2018.

[45] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems (TOCS)*, 1(1):3–23, 1983.

[46] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computer Surveys*, 22(4), Dec. 1990.

[47] ScyllaDB. ScyllaDB :: Scylla Is Next Generation NoSQL. `http://www.scylladb.com/`, 2018.

[48] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM Symposium on Operating System Principles (SOSP)*, Nov 2013.

[49] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[50] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec 2002.

[51] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *ACM Symposium on Operating System Principles (SOSP)*, Oct 2015.

[52] F. Zhou, Y. Gan, S. Ma, and Y. Wang. wPerf: generic Off-CPU analysis to identify bottleneck waiting events. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2018.

A Artifact Appendix

A.1 Abstract

This appendix presents the steps for installing Eiger-PORT and running experiments that compare the performance of Eiger-PORT and its base system, Eiger. Eiger-PORT is implemented as a modification to Eiger's code base, which is built on top of Cassandra and written in Java. The experiments evaluate latency, throughput, and scalability. The results are expected to show that Eiger-PORT outperforms Eiger in all experiments and the performance advantages become more significant under more skewed workloads. Eiger-PORT's better performance comes from its performance-optimal read-only transactions.

A.2 Artifact check-list

- **Hardware:** 2.4 GHz Quad-Core Xeon CPU, 12 GB RAM, 1 Gbps network interface
- **Metrics:** latency, throughput, scalability
- **Expected experiment run time:** 10–20 hours
- **Public link:** `http://github.com/princeton-sns/Eiger-PORT.git`

A.3 Description

A.3.1 How to access

The code base of Eiger-PORT is publicly accessible on Github at `http://github.com/princeton-sns/Eiger-PORT.git`. It includes a README file that provides step-by-step instructions on how to set up the environment and run experiments.

A.4 Installation

Please clone the code repository under a clean directory on a machine. The scripts in the package will work seamlessly if the repository is cloned under */local*. The required dependencies can be installed by simply running the bash file *install-dependencies.bash*. Apache Ant is used to build the source code. Both the system files and the stress tool need to be compiled. Please see the README file in the repository for more details.

A.5 Experiment workflow

Running experiments as described in the paper requires setting up two clusters with each having 8 servers and 8 clients. One cluster is the active cluster for processing transactions and the other cluster is used as a replica, which passively receives replicated writes from the active cluster. One extra machine is needed for the control node. Therefore, to create an 8-server-8-client environment, 33 machines are needed in total (2 clusters, 16 machines in each, and 1 control node).

When the experiment topology is determined, the configuration files under the directory *vicci_dcl_config* need to be modified accordingly. All the scripts used to run experiments are under the directory *eval-scripts*. Experiments can be launched by executing *latency_throughput.bash*. The experimental parameters, such as Zipfian constant and read-to-write ratio, are specified in the file *dynamic_defaults*. For details, please see the README file.

A.6 Evaluation and expected result

The results of each experiment are stored under the directory *experiments/dynamic*. Throughput numbers are shown in the file *combined.graph*. A set of latency processing scripts are provided under the directory *data_proc_scripts*. Eiger-PORT is expected to have ~2*X* higher throughput and ~50% latency compared to Eiger. The performance advantages of Eiger-PORT are expected to become more significant under more skewed workloads.

A.7 AE Methodology

Submission, reviewing and badging methodology:

- `https://www.usenix.org/conference/osdi20/call-for-artifacts`



Toward a Generic Fault Tolerance Technique for Partial Network Partitioning

Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, Samer Al-Kiswany
University of Waterloo, Canada

Abstract

We present an extensive study focused on partial network partitioning. Partial network partitions disrupt the communication between some but not all nodes in a cluster.

First, we conduct a comprehensive study of system failures caused by this fault in 12 popular systems. Our study reveals that the studied failures are catastrophic (e.g., lead to data loss), easily manifest, and can manifest by partially partitioning a single node.

Second, we dissect the design of eight popular systems and identify four principled approaches for tolerating partial partitions. Unfortunately, our analysis shows that implemented fault tolerance techniques are inadequate for modern systems; they either patch a particular mechanism or lead to a complete cluster shutdown, even when alternative network paths exist.

Finally, our findings motivate us to build Nifty, a transparent communication layer that masks partial network partitions. Nifty builds an overlay between nodes to detour packets around partial partitions. Our prototype evaluation with six popular systems shows that Nifty overcomes the shortcomings of current fault tolerance approaches and effectively masks partial partitions while imposing negligible overhead.

1 Introduction

Modern networks are complex. They use heterogeneous hardware and software [1], deploy diverse middleboxes (e.g., NAT, load balancers, and firewalls) [2, 3, 4], and span multiple data centers [2, 4]. Despite the high redundancy built into modern networks, catastrophic failures are common [1, 3, 5, 6]. Nevertheless, modern cloud systems are expected to be highly available [7, 8] and to preserve stored data despite failures of nodes, networks, or even entire data centers [9, 10, 11].

We focus our investigation on a peculiar type of network fault: *partial network partitions*[1], which disrupts the communication between some, but not all, nodes in a cluster. Figure 1 illustrates how a partial network partition divides a cluster into three groups of nodes, such that two groups (Group 1 and Group 2) are disconnected, but Group 3 can communicate with Groups 1 and 2.

In our previous work [12] we identified this fault and presented examples of how it leads to system failures. Other than our previous preliminary effort, we did not find any in-depth analysis of partial network partition failures and of their fault tolerance techniques. Nevertheless, we found 51 reports of failures caused by partial network partitioning faults[2] in the publicly accessible issue tracking systems of 12 production-quality systems (Section 4), numerous blog posts and discussions of this fault on developers' forums (Section 3), and eight popular systems with fault tolerance techniques specifically designed to tolerate this type of fault (Section 5).

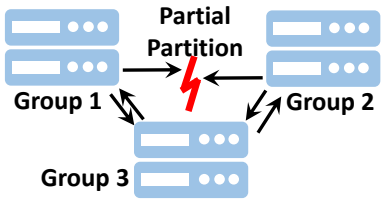


Figure 1: Partial partition. Groups 1 and 2 are disconnected, while Group 3 can reach both sides of the partition.

Our goal in this work is threefold. First, we aim to study failures caused by partial network partitioning to understand their impact and failure characteristics and, foremost, to identify opportunities to improve systems' resiliency to this type of fault. Second, we aim to dissect the fault tolerance techniques implemented in popular production systems and identify their shortcomings. Third, we aim to design a generic fault tolerance technique for partial network partitioning. This is the first work to characterize these failures and explore fault tolerance techniques for partial partitioning faults.

It is important to understand that *partial* partitions are fundamentally different from *complete* partitions [12]. Complete partitions split a cluster into two completely disconnected sides and are well studied with known theoretical bounds (CAP theorem [13]) and numerous practical solutions [14, 15, 16, 17]. On the contrary, a cluster experiencing a partial partition is still connected but not all-to-all connected. Consequently, the theoretical bounds of complete partitions do not apply to partial partitions, and fault tolerance techniques for complete partitions are not effective in handling partial partitions (Section 8).

An analysis of partial network partitioning failures. We conduct an in-depth study of 51 partial network partitioning failures from 12 cloud systems (Section 4). We select a diverse set of systems, including database systems (MongoDB and HBase), file systems (HDFS and MooseFS), an object storage system (Ceph), messaging systems (RabbitMQ, Kafka, and ActiveMQ), a data-processing system (MapReduce), a search engine (Elasticsearch), and resource managers (Mesos and DKron). For each considered failure, we carefully study the

[1] This is the commonly used name in failure reports and discussion forums.

[2] A *fault* is the initial root cause. If not properly handled, it may lead to a user-visible system *failure*.

failure report, logs, discussions between users and developers, source code, and code patches.

Failure Impact. Overall, we find that partial network partitioning faults cause silent failures with catastrophic effects (e.g., data loss and corruption) that affect core system mechanisms (e.g., leader election and replication).

Ease of manifestation. Unfortunately, these failures can easily occur. The majority of the failures are deterministic and require less than four events (e.g., read or write request) for the failure to occur. Even worse, all the studied failures can be triggered by partially partitioning a single node. The majority of failures do not require client access or can be triggered by clients only accessing one side of the partition.

Insights. We identify three approaches to improve system resilience: better testing, focused design reviews, and building a generic fault tolerance communication layer. Our analysis of each failure's manifestation sequence, access patterns, and timing constraints shows that almost all the failures could have been revealed through simple tests and by only using five nodes. Second, the majority of failures are due to design flaws. We posit that design reviews focused on network partitioning could identify these vulnerabilities. Third, building a generic communication layer to mask partial partitions is feasible, simplifies system design, and improves system resiliency.

Finally, we identify that a common deployment approach of Zookeeper introduces a failure vulnerability (Section 5). Our analysis shows that system designers need to design additional mechanisms to handle partial partitions when using Zookeeper or other external coordination services.

Dissecting modern fault tolerance techniques. We dissect the implementation of eight popular systems (VoltDB, MapReduce, HBase, MongoDB, Elasticsearch, Mesos, LogCabin, and RabbitMQ) and study the fault tolerance techniques they employ specifically to tolerate partial partitions (Section 5). For each system, we study the source code, analyze the fault tolerance technique's design, extract the design principles, and identify the technique's shortcomings. We identify four principled approaches for tolerating partial partitions: identifying the surviving clique, checking neighbors' views, verifying failures announced by other nodes, and neutralizing partially partitioned nodes.

Our analysis reveals that the studied fault tolerance techniques are inadequate. They either patch a specific system mechanism, which leaves the rest of the system vulnerable to failures, or unnecessarily shut down the entire cluster or pause up to half of the cluster nodes (Section 5).

Designing a generic fault tolerance technique. Our findings motivate us to build the **n**etwork part**i**tioning **f**ault-**t**olerance la**y**er (Nifty), a simple, generic, and transparent communication layer that can mask partial network partitions (Section 6). Nifty's approach is simple; it monitors the connectivity in a cluster through all-to-all heart beating, and when it detects a partial partition, it detours the traffic around the partition through intermediate nodes. Nifty overcomes all the shortcomings present in the studied fault tolerance techniques.

The main insight of Nifty is that tolerating partial partitioning does not require elaborate techniques such as the ones adopted by current systems (Section 5). Many modern systems already incorporate membership and connectivity monitoring mechanisms based on all-to-all heart beating [18, 19, 20]. Nifty shows that extending these mechanisms with a simple rerouting capability can effectively mask partial partitions.

To demonstrate Nifty's effectiveness, we deploy it with six systems: HDFS, Kafka, RabbitMQ, ActiveMQ, MongoDB, and VoltDB. We choose these systems because they are data intensive and popular systems. Furthermore, RabbitMQ and VoltDB implement generic techniques to tolerate partial partitions. Our prototype evaluation with synthetic and real-world benchmarks shows that Nifty effectively masks partial partitions while adding negligible overhead.

2 Definitions

A *partial network partition* is a network fault that prevents at least one node (e.g., a node in Group 1 in Figure 1) from communicating with another node (Group 2) in the system, while a third node (Group 3) can communicate with both affected nodes. Nodes in a partially partitioned cluster are still connected but are not all-to-all connected (i.e., they do not form a complete graph [21]). A partial partition divides a cluster into three groups: two sides and one bridge group. We identify a node as a *bridge* node if it can reach at least one node on each side of a partition. A partial partition has two *sides*, all the nodes on one side of the partition cannot reach all the nodes on the other side of the partition. We note that a cluster may suffer from multiple concurrent partial partitions.

We define a *single-node partial partition* as a partial partition that has a single node on one side of the partition, while the rest of the cluster nodes are bridge nodes or are on the other side of the partition. For instance, a single-node partial partition can be caused by a firewall misconfiguration that prevents a node from communicating with some other nodes.

3 Causes of Partial Network Partitioning

Recent reports indicate that network partitioning faults are common and happen at various scales. Connectivity loss between data centers [1] leads to network partitions in geo-replicated systems. Wide area network partitions happen as frequently as once every four days [6]. Switch failures can cause a network partition in a data center [5]. Switch failures caused 40 network partitions in two years at Google [3] and 70% of the downtime at Microsoft [5]. On a single node, NIC [22] or software failures can partition a node that may host multiple VMs. Finally, network partitions caused by cor-

Table 1: List of studied systems and the number of studied failures. The shaded rows are systems that implemented a fault tolerance technique for partial network partitioning.

System	Category	Failures	
		Total	Catastrophic
Elasticsearch [32]	Search engine	17	17
MongoDB [33]	Database	9	5
RabbitMQ [18]	Messaging	5	3
MapReduce [34]	Data processing	4	2
HBase [35]	Database	3	2
Mesos [36]	Resource mngr.	2	1
HDFS [34]	File system	3	1
Ceph [20]	Storage system	2	2
MooseFS [37]	File system	2	2
Kafka [38]	Messaging	2	2
ActiveMQ [39]	Messaging	1	1
DKron [40]	Resource mngr.	1	1
Total	**-**	**51**	**39**

related failures are common [4, 5, 6] and often caused by system-wide maintenance tasks [3, 5].

We found 51 failure reports detailing system failures due to partial network partitions, and numerous articles and online discussions discussing the fault [23, 24, 25, 26]. Some of these reports and discussions mention the root cause of the partial partition. Partial partitions are caused by a connectivity loss between two data centers [1] while both are reachable by a third center, the failure of additional links between racks [27, 28], network misconfiguration [29], firewall misconfiguration [29], network upgrades [30], and flaky links between switches [31]. Unfortunately, we did not find failure reports that detail partial partitioning faults in production networks.

4 Analysis of Partial Network-Partitioning Failures

We conduct an in-depth study of partial network partitioning failures reported in 12 popular systems (Table 1). We aim to understand the impact and characteristics of these failures and to identify opportunities for improving system resilience.

4.1 Methodology

We choose 12 diverse and widely used systems (Table 1), including two databases, a data analysis framework, two file systems, three messaging systems, a storage system, a search engine, and two distributed resource managers.

We selected the 51 failures in our study from publicly accessible issue-tracking systems. First, we used the search tools in the issue-tracking systems to find tickets related to partial network partitioning. Users did not classify network partitioning failures based on the partition type, so we had to search for all network partitioning failures and manually identified partial partitioning failures. We used the following keywords: "network partition," "partial network partition," "partial partition," "network failure," "switch failure," "isolation," "split-brain," and "asymmetric partition." Second, we considered tickets that were dated 2011 or later. Third, we excluded tickets marked as "Minor." For each ticket, we studied the failure description, system logs, developers' and users' comments, and code patches. For tickets that lacked enough details (e.g., missing output logs or did not have details about the affected mechanism), we manually reproduced them using NEAT [12]. Finally, during our evaluation, we found and reported bugs in Kafka and Elasticsearch. We included these failures in our study.

We differentiate failures by their manifestation sequences. In a few cases, the same faulty mechanism leads to two different failure paths. We count these as separate failures, even if they are reported in a single ticket. Similarly, although the exact failure is sometimes reported in multiple tickets, we count it once in our study.

4.2 Limitations

As with any characterization study, our findings may not be generalizable. Here, we list four potential sources of bias and describe our best efforts to address them.

1. *Representativeness of the studied systems*. Although we study 12 diverse systems (Table 1), our results may not be generalizable to systems we did not study. The selected systems follow diverse designs from strongly consistent (MongoDB, HBase, and Ceph) to eventually consistent (Elasticsearch) designs and from systems persisting data on disks and replicating data in-memory across nodes to caching systems. They follow a primary-backup or peer-to-peer architecture and use synchronous or asynchronous replication. The selected systems are widely used: Kafka, ActiveMQ, and RabbitMQ are the most popular open-source messaging systems; MapReduce, HDFS, and HBase are the core of the Hadoop platform; Elasticsearch is a popular search system; and MongoDB is a popular database.

2. *Limited number of tickets*. We study all 51 tickets that we found following our methodology. Statistical inference indicates that 30 samples can sufficiently represent the entire population [41]. More rigorously, if we assume the tickets we found represent a random sample of partial network partition failures in the wild, the central limit theorem predicts that our analysis of 51 tickets has a 13% margin of error at a 95% confidence level. To increase confidence in our findings, we only report findings that apply to at least two-thirds of the studied failures. A third of our findings apply to all failures.

Table 2: Failure impact and percentages of how many failures caused the corresponding impact.

Impact	%	
Data loss	23.5%	Catastrophic (74.5%)
System unavailability	21.6%	
Stale read	15.7%	
Data corruption	5.9%	
Dirty read	3.9%	
Data unavailability	3.9%	
Reduced availability	23.5 %	
Other	2%	

3. *Priority bias*. We include only high-impact tickets and avoid tickets marked by the developers as low-priority. This sampling methodology may bias the results.

4. *Observer error*. To reduce the chance for observer errors, two team members study every failure report using the same classification methodology. Then, we discuss the failure in a group meeting before reaching a verdict.

4.3 Findings

This subsection presents nine general findings. Our study indicates that partial network partitioning leads to catastrophic, silent failures. Surprisingly, these failures are easy to manifest. The majority of failures are deterministic, require a single-node partial partition, and require a few events to manifest. However, our study also identifies failure characteristics that can inform system designs and improve testing. Finally, we find that the majority of the studied failures are due to design flaws, indicating that developers do not expect networks to fail in this way.

Finding 1: *A significant percentage (74.5%) of the studied failures have a catastrophic impact.*
A failure is said to be catastrophic if it leads to a system crash or violates the system's guarantees (Table 2). Failures that reduce availability (e.g., crash of a single replica) or degrade performance are not considered catastrophic.

Data loss is the most common impact of partial network failures. For instance, in HBase, region servers store their logs on HDFS. When a log reaches a certain size, the region server creates a new log and informs the master of the new log location. If a partial partition isolates a region server from the master while both can reach HDFS, the master assumes that the region server has failed and assigns its logs to a new region server. If the old region server creates a new log, the master will not know about it, and the entries in the new log will be lost [42].

The second most common catastrophic impact of partial partitions is complete cluster unavailability, from which the majority of the studied systems suffer. A glaring example of this failure is the common deployment approach of Zookeeper. For instance, in ActiveMQ, a ZooKeeper service [43] monitors the cluster master and selects a new master if the current one fails. If a partial partition isolates the master from all ActiveMQ nodes while all nodes are reachable from ZooKeeper, the nodes will pause their operations because they cannot reach the master. Because ZooKeeper can reach the current master, it neither detects the problem nor selects a new master. The cluster remains unavailable until the partial partition heals [44]. Kafka and Mesos use Zookeeper in a similar fashion and suffer from a similar failure. The rest of the catastrophic failures lead to stale reads, data corruption, loss of data availability, and dirty reads.

In 23.5% of the failures, a partial partition unnecessarily reduces system availability. For example, leader election in MongoDB is based on a majority vote, with an arbiter node included to break ties. Unfortunately, this design leads to cluster unavailability under partial network partitions. For instance, consider a shard that has two replicas (A and B), with A being the leader. If a partial partition disrupts the communication between A and B while both can reach an arbiter, B will detect that A is unreachable and calls for a leader election. Because there is only one candidate in the system, the arbiter votes for it, and B becomes the leader. The arbiter will inform A of the new leader, and A steps down. A will detect that the leader (B) is unreachable, call for a leader election, become a leader, and then B steps down. This leader-election thrashing continues until the network partition heals [45]. The system is unavailable during leader election, so this failure significantly reduces system availability. We discuss the resolution of this failure in Section 5.

Finding 2: *Most of the studied failures (84.3%) are silent — the user is not informed about their occurrence.*
Despite the dangerous impact of partial partitioning faults, most systems do not report to the user that a failure has occurred. This is unsettling because a lack of error or warning notification delays failure detection. Some systems return a warning to the user when an operation fails due to partial network partitioning, but these warnings are ambiguous with no clear mechanisms for resolution. For example, in Elasticsearch, if a client sends a request to a replica that is partially isolated from the other replicas, the replica will return "a rejected execution" exception [46]. This confusing warning does not inform the user of the problem's actual cause nor the steps needed to resolve it.

Finding 3: *Leader election, configuration change, and replication protocol are the mechanisms most vulnerable to partial network partitioning.*
Leader election is the mechanism most vulnerable to partial network partitions (Table 3). In most cases, these failures lead to electing two leaders, one at each side of the partition [47, 48].

Configuration change is the second-most affected mechanism. For instance, each node in RabbitMQ maintains a membership log that lists the current nodes in the cluster. If

Table 3: Failure percentages per affected mechanism.

Mechanism	%
Leader election	37.3%
Configuration change	19.6%
Replication protocol	17.6%
Request routing	11.8%
Scheduling	5.9%
Data migration	5.9%
Data consolidation	2%

nodes have conflicting views on which nodes are part of the cluster, the RabbitMQ cluster crashes. For instance, in a cluster with three nodes (A, B, and C), when a partial partition disconnects B and C, B assumes that C crashed and removes it from the membership log, and C assumes that B crashed and removes it from the membership log. This inconsistency in the cluster membership leads to a complete cluster crash [49].

The replication mechanism is the third-most affected mechanism. For instance, if a partial partition in Elasticsearch isolates a shard's leader from the majority of that shard's replicas, the leader will wait for a period of time before stepping down. In this period, the leader continues to accept client write operations and acknowledges them before successfully replicating them [50]. If a client writes to the leader and later reads from one of the other replicas, it may read stale data.

Finding 4: *Most failures (60.8%) do not require client access or require only that clients access one side of the partition.*
To reduce the network partition's impact, some systems limit client access to one side of the partition [51, 52, 53]. However, our analysis shows that 60.8% of failures require no client access at all or only client access to one side of the partition. As an example of a failure that does not require client access, in MongoDB, balancer servers monitor the cluster load and migrate data chunks between nodes to rebalance the load across nodes. After rebalancing the data, a balancer updates Mongo's metadata server with the new data location. If during a re-balance operation of a particular shard a partial partition isolates the balancer from the metadata service, the cluster metadata will be in an inconsistent state, leading to the unavailability of that shard [54].

This finding highlights that system designers should consider the impacts of partial partitioning faults on all operations, including background operations.

Finding 5: *The majority of failures (68.6%) require three or fewer events (other than the partial partition) to manifest.*
Only a few events need to occur for a failure to happen. An event is a user request, a hardware or software fault, or a start of a background operation (e.g., leader election and data rebalancing). This is alarming because a small number of events can lead to catastrophic failures. Especially that in real deployments, many users interact with the system, increasing the probability of failure. Table 4 shows that, in 13.7% of failures, a partial partition, without any additional events, leads to a failure.

Table 4: Number of events required for a partial network partitioning failure to manifest.

Number of events	%
1 (Just a partial partition)	13.7%
2	9.8%
3	31.4%
4	13.7%
>4	31.4%

Table 5: System connectivity during a partial partition.

Network Partition Characteristic		%
Partition any node		33.3%
Partition a specific node		66.6%
• Leader	45.1%	
• Nodes with a special role	9.8%	
• A central service	7.8%	
• New nodes	2%	

Finding 6: *All the studied failures can be triggered by a single-node partial partition, with 33.3% of them happen by partitioning any node.*
Arguably, single-node partial partitions (Section 2) are generally more likely than partitioning more than one node. These partitions could happen due to a single ToR switch malfunction or by misconfiguring a single machine's firewall.

We further study which nodes need to be isolated for a failure to manifest (Table 5). Of the failures, 33.3% manifest by partitioning any node in the system—regardless of its role. Among the failures that require partitioning a specific node, partitioning the leader replica is most common (45.1%). In real deployments, partitioning a leader is likely because almost every node in the cluster is a leader for some shard. Partitioning a node with a special role (such as an arbiter in MongoDB) causes 9.8% of the failures.

Finding 7: *All the studied failures, except one, are deterministic or have known time constraints.*
Table 6 shows the timing constraints of the studied failures. Almost all the failures are either deterministic with no timing constraints (i.e., whenever the event sequence happens, a failure happens) or have known timing constraints, such as the period before considering a node to have failed. Only one failure is nondeterministic, as an interleaving of multiple threads causes it.

Table 6: Failures' timing constraints.

Timing constraint	%
No timing constraints	64.7%
Known timing constraints	33.3%
Nondeterministic	2%

Table 7: Percentage of design and implementation flaws.

Flaw type	%	Average Time to Resolution
Design	41.2%	260 days
Implementation	31.4%	98 days
Unresolved	27.5%	-

Table 8: Number of nodes needed to reproduce a failure.

Number of nodes	%
3 nodes	76.5%
4 nodes	21.6%
5 nodes	2%

Finding 8: *The resolution of 56.8% of the fixed failures required changing the design of a protocol or a mechanism.* We consider a code patch to be fixing a design flaw if it significantly changes the implemented protocol or logic, such as changing the mechanism to select a master in Elasticsearch.

Most of the fixed failures are caused by a design flaw (Table 7). This indicates that system designers overlook partial network partitioning failures in the design phase. We argue that a design review focused on partial partitions would detect a system's vulnerability to these failures.

Finding 9: *All failures can be reproduced with five nodes, and all but one can be reproduced using a fault injection tool.* These failures can be easily reproduced with small clusters of five or fewer nodes (Table 8), and 76.5% require only three nodes. Furthermore, all the failures except one can be reproduced using a fault-injection framework that can inject partial partitioning faults such as NEAT [12].

4.4 Insights

Our analysis shows that partial partitions lead to catastrophic silent failures that are easy to manifest, are deterministic, and can be triggered by a single-node partial partition and a sequence of a few events.

Fortunately, we identify three approaches for improving system resilience to partial partitions. First, because these faults are deterministic and can be reproduced on a small cluster, improved testing can reveal the majority of the studied failures. Our analysis finds timing, client access, and partition characteristics that significantly reduce the number of sufficient test cases. Second, our study of the code patches reveals that focused design reviews can identify system vulnerabilities early in the design process.

Third, partial network partitions have two characteristics that imply that a generic fault tolerance technique is possible. These faults can be detected by exchanging information between the nodes, and by definition, there are alternative paths in the network to reconnect the system. We leverage these two characteristics in building Nifty (Section 6).

Most of the studied failures are caused by the underlying assumption that, if a node can reach a service, all nodes can reach that service, and if a node cannot reach a service then the service is down. Our analysis shows the danger of such assumptions; this leads to a confusing state, wherein some of the system's parts start executing a fault tolerance mechanism, while others presume the whole system is healthy and carry on normal operations. The mix of these two operation modes is poorly understood and tested.

Finally, we identify that a common usage of external coordination services (e.g., Zookeeper) introduces a vulnerability to partial network partitioning fault. System designers need to build additional techniques to detect and handle partial partitions when using external coordination services.

5 Dissecting Modern Fault Tolerance Techniques

We studied the code patches related to the tickets included in our study. Six of the systems in Table 1 (MongoDB, Elasticsearch, RabbitMQ, HBase, MapReduce, and Mesos) changed the system design to incorporate a fault tolerance technique specific to partial network partitioning faults. The rest of the systems either patched the code with an implementation-specific workaround or did not fix the reported bugs yet.

Furthermore, we found that two additional systems, VoltDB [19, 55] and LogCabin [56] (the original implementation of the Raft [14] consensus protocol), implement fault tolerance techniques for partial partitions. For these two systems, we did not find failure reports related to partial partitioning faults in their issue tracking systems, but VoltDB announced that their recent version tolerates partial partitions [57]. We experimented with LogCabin to understand the impact partial partitions have on strongly consistent systems and found that LogCabin incorporates a technique to tolerate partial partitions. We included VoltDB and LogCabin in our study.

For each of the eight systems, we study the source code, and extract and analyze the design principles of their fault tolerance technique. We identify four approaches for tolerating partial partitions: detecting a surviving clique of nodes, checking neighbors' views, verifying failure reports received from other nodes, and neutralizing one side of the partial partition. Unfortunately, these techniques have severe shortcomings that may lead to a complete system shutdown or to the unavailability of a major part of the system. In this section, we detail these techniques and discuss their shortcomings.

5.1 Identifying the Surviving Clique

Main idea. Upon a partial network partition, the system identifies the maximum clique of nodes [58], which is the largest subset of nodes that are completely connected. All nodes that are not part of the maximum clique are shut down. VoltDB follows this approach.

VoltDB Implementation. VoltDB [19, 55] is a popular ACID, sharded, and replicated relational database. VoltDB

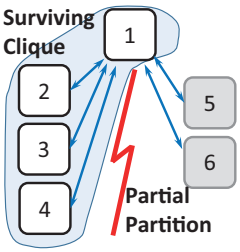


Figure 2: VoltDB's surviving clique. Gray nodes shut down as they are not in the clique.

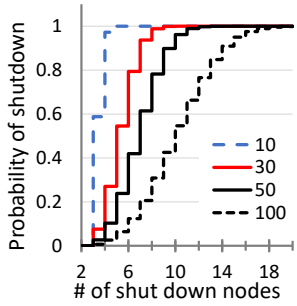


Figure 3: The probability of a VoltDB cluster shutdown. Different lines represent different cluster sizes. The x-axis shows the number of nodes that are not in the clique.

follows a peer-to-peer approach to implement this technique. Every node in the system periodically sends a heartbeat to all nodes. If a node loses connectivity to any node, it suspects that a partial network partition occurred and starts the recovery procedure. The recovery procedure has two phases. In the first phase, the node that detects the failure broadcasts a list of nodes it can reach. When a node in the cluster receives this message, it broadcasts its list of reachable nodes to all nodes in the cluster. In phase two, every node independently combines the information from the other nodes into a graph representing the cluster connectivity. Each node analyzes this graph to detect the maximum completely connected clique of nodes. Every node that finds that it is not part of this "surviving" clique shuts itself down. Figure 2 shows an example in which a partial partition disrupts the communication between nodes 2, 3, and 4 on one side and nodes 5 and 6 on another. Nodes 5 and 6 are not part of the clique and will shut down.

After identifying the surviving clique, the system verifies that it did not lose any data by verifying that the surviving clique has at least one replica of every data shard. If the clique is missing one shard, such as when all the replicas of a shard are shut down, the entire system shuts down.

Shortcomings. This fault tolerance approach has two severe shortcomings. First, it unnecessarily shuts down up to half of the cluster nodes, reducing the system's performance and fault tolerance. Second, this approach causes a complete cluster shutdown if the surviving clique is missing a single data shard. To understand how likely a cluster is to shut down, we conduct a probabilistic analysis (detailed in our technical report [59]). Figure 3 shows the probability of a complete cluster shutdown while varying the cluster size and the number of nodes that shut down (i.e., nodes that are not part of the surviving clique – the x-axis in Figure 3). Each shard has three replicas. Our analysis shows that isolating only 10% of the nodes leads to more than a 50% probability of shutting down the entire cluster, and isolating only 20% of the nodes leads to a staggering 90% chance of a complete cluster shutdown.

5.2 Checking Neighbors' Views

Main idea. When one node (e.g., node S) loses its connection to another node D, it verifies whether the connection is lost due to a partial partition. To this end, S asks all nodes in the cluster whether they can reach D. If a node reports that it can reach D, this indicates that the cluster is suffering a partial network partition.

If S detects a partial network partition, S either disconnects from all nodes that can reach D, which effectively makes the partition a complete partition, or pauses its operation. RabbitMQ and Elasticsearch follow this approach.

5.2.1 RabbitMQ

RabbitMQ [18] is a popular messaging system that replicates message queues for reliability. In RabbitMQ, if a node detects that its communication with another node (e.g., node D) is affected by a partial partition, it applies one of the following policies depending on its configuration.

1. *Escalate to a complete partition.* The node will drop its connection with any node that can reach node D. The goal of this policy is to create a complete partition in which both sides work independently. This configuration leads to data inconsistency and requires running a data consolidation mechanism after the partition heals.

2. *Pause*: To avoid data inconsistency, once a node discovers the partial partition, it pauses its activities. It resumes its activities only when the partition heals. The result of this policy is that a subset of nodes will continue to operate. This subset will be completely connected and will run without sacrificing data consistency.

3. *Pause if anchor nodes are unreachable*: RabbitMQ's configuration can specify a subset of nodes to act as anchor nodes. If a node cannot reach any of the anchor nodes, it pauses. This may lead to creating multiple complete partitions if the anchor nodes become partially partitioned. This may lead to pausing all nodes if all the anchor nodes are isolated.

After a partition heals, RabbitMQ employs two data consolidation techniques: administrator intervention, in which the administrator decides which side of the partition should become the authoritative version of the data, and auto-heal, in which the system makes this determination based on the number of clients connected to each side. Both techniques may lead to data loss or inconsistency [12].

Shortcomings. RabbitMQ's policies have serious shortcomings. Changing a partial partition to a complete partition (policies 1 and 3) may lead to multiple inconsistent copies of the data, whereas the *pause* policy (policy 2) may pause the entire system or the majority of the nodes. For instance, in Figure 4, if every node except node 1 detects that it cannot reach a

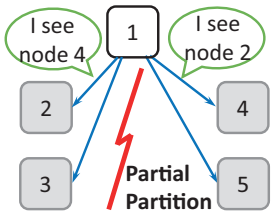


Figure 4: A scenario for RabbitMQ's *pause* policy. Every non-bridge node pauses (gray nodes) as it detects that it cannot reach one node on the other side.

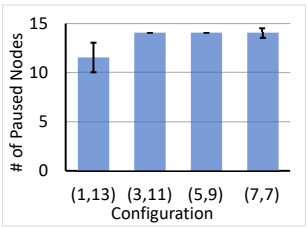


Figure 5: The median number of paused nodes in a cluster of 15 nodes. In all runs, one node is unaffected by the partition. The notation (i, j) shows the number of nodes on each side of the partition.

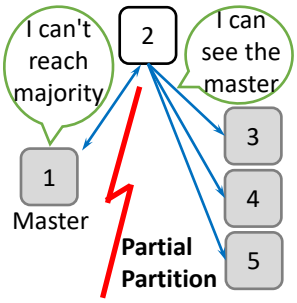


Figure 6: Elasticsearch unavailability scenario. The master pauses because it cannot reach majority of nodes, and all nodes pause because they cannot reach the master.

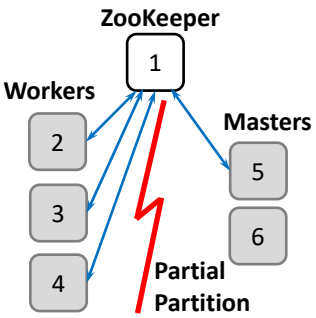


Figure 7: A Mesos cluster becomes unavailable when a partial partition isolates the master node and its backups.

node on the other side of the partition, it pauses, leading to a complete cluster pause.

In the case of the *pause* policy (policy 2), to determine how many nodes pause under different partial partition scenarios, we conduct an experiment in which we deploy a 15-node RabbitMQ cluster, introduce a partial partition, and observe how many nodes pause. In all experiments, we inject a partition such that one node remains unaffected and able to reach all nodes. Figure 5 shows the median number of paused nodes under various partition configurations. We run each configuration 30 times. Surprisingly, in all configurations almost all the cluster nodes pause because each node detects that it cannot reach at least one node on the other side of the partition. Even isolating a single node (configuration (1,13) in Figure 5) leads to pausing 12 nodes. Our investigation reveals that nodes declare another node unreachable after missing its heartbeats for a timeout period. In RabbitMQ, the default timeout period is 1 minute, which gives enough time for many nodes to detect the partition and pause. Using a shorter timeout periods causes some nodes to declare prematurely that other nodes have failed, even without a partial partition.

5.2.2 Elasticsearch

Elasticsearch [32] is a popular search engine. Its master election protocol uses a fault tolerance technique based on checking neighbors' views. In Elasticsearch, the node with the lowest ID is the master. If a node (e.g., S) cannot reach the master, it contacts all nodes to check whether they can reach the master. If any node reports that it can reach the master, S pauses its operations. If none of the nodes can reach the master, the node with the lowest ID becomes the new master.

Shortcomings. First, this approach can affect cluster availability quite severely, as all nodes that cannot reach the master pause. In the worst case, it can cause a complete cluster unavailability. For instance, in Figure 6, none of the nodes can reach the master except node 2, which refuses to become the new master because it can reach a node with a lower ID (node 1). Consequently, all the nodes in the cluster pause. Furthermore, because the master cannot reach a majority of nodes, it also pauses, which leads to system unavailability [60]. Second, Elasticsearch uses this approach only to fortify the master election protocol, which leaves the rest of the system vulnerable to partial partitions.

5.3 Failure Verification

Main idea. If a node (e.g., S) receives a notification from another node that a third node (D) has failed, node S first verifies that it cannot reach D before taking any fault tolerance steps. This approach is used in the leader election protocols of MongoDB [33] and LogCabin [56]. It was also used in an earlier version of Elasticsearch.

In MongoDB and LogCabin, if a leader is on one side of a partial partition but can still reach the majority of nodes, the nodes on the other side of the partition unnecessarily call for leader election. Finding 1 in Section 4 discusses a scenario in which a partial partition leads to continuous leader election thrashing and to system unavailability [45]. To avoid unnecessary elections, when a node receives a call for election, it first verifies that the current leader is unreachable. A node participates in an election only if it cannot reach the current leader, else it will ignore the failure report.

Shortcomings. This approach has two major shortcomings. First, it leads to the unavailability of a large number of nodes. Second, it is mechanism specific. Designing a system-wide fault tolerance mechanism using this approach is tricky because one cannot ignore every failure notification. For instance, using this approach in an earlier version of Elasticsearch backfired [61]. During data migration from a primary replica of a shard to a secondary replica, if a partial partition isolates the primary replica from the secondary replica while both are reachable from the master node, the primary requests a new secondary replica. Because the master can reach the secondary replica, it ignores the failure report. This leads to the unavailability of the affected shard [61]. Broadly applying

Table 9: Summary of shortcomings. (D) indicates that the nodes shut down. (P) indicates that the nodes pause until the partition heals. In the worst case, RabbitMQ pauses all nodes except one. We consider this a complete cluster loss (1). Under different RabbitMQ policies, (2) and (3) can occur. (S) indicates a system-wide technique, whereas (M) is a mechanism-specific technique.

	Surviving Clique	Checking w/ Neighbors		Failure Verification	Neutralizing Nodes		Nifty
	VoltDB	Elasticsearch	RabbitMQ	MongoDB/LogCabin	MapReduce/HBase	Mesos	
Reduced Availability	$\times^{D}$	$\times^{P}$	$\times^{P}$	$\times^{P}$	$\times^{D}$	$\times^{P}$	
Complete Unavailability	$\times$	$\times$	$\times^{1}$				
Complete Partition			$\times^{2}$				
Double Execution						$\times$	
Data Unavailability			$\times^{3}$				
Scope (System/Mechanism)	S	M	S	M	M	M	S

this fault tolerance technique is not feasible because designers have to revisit the design of every system mechanism, consider the consequences of ignoring failure reports, and examine the interaction of various mechanisms under partial partitions.

5.4 Neutralizing Partitioned Nodes

Main idea. One challenge related to handling partial network partitions is that nodes may update a shared state that is reachable from both sides of the partition, leading to data loss and inconsistency. To avoid this problem, this approach attempts to neutralize one side of the partition. However, the neutralization method is implementation-specific. HBase, MapReduce, and Mesos use this approach.

HBase Implementation. In HBase, data shards are managed by an HBase node but are stored on HDFS. If the HBase leader cannot reach one of the HBase nodes, it neutralizes that node by renaming the shard's directory in HDFS. Renaming a shard's directory effectively prohibits the old HBase node from making further changes to the shard [42]. The leader then assigns the shards of that node to a new HBase node.

MapReduce Implementation. In MapReduce, a manager node assigns tasks to AppMaster nodes. If the manager cannot reach an AppMaster, it reschedules the tasks assigned to that AppMaster to a new AppMaster. With partial network partitions, this approach may result in two AppMasters working on the same task, which leads to data corruption [62]. To fix this problem, when an AppMaster completes a task, it writes a completion record in a shared log on HDFS. Before an AppMaster executes a new task, it checks the shared log for a completion record. If it finds one, it does not re-execute the task.

Mesos Implementation. In Mesos, a master node assigns tasks to worker nodes. A Zookeeper instance selects the master node. The master sends periodic heartbeats to workers. If a partial partition isolates a worker node from the master, it pauses its operations. Figure 7 shows a worst-case scenario in which the partial partition isolates the master and its backup from all workers, which leads to a complete cluster unavailability. Finally, if a master detects that one of the workers is unavailable, it marks the tasks that were running on the unreachable worker as lost and reschedules them on new workers. This may lead to the double execution of a task [63].

Shortcomings. First, it is not practical to use this approach for system-wide fault tolerance, as this approach is specific to a certain protocol and implementation. The presented three systems use this approach for different mechanisms. To use this approach broadly, designers must go through the daunting task of independently designing a fault tolerance technique for every mechanism in the system and understanding the interaction between these mechanisms. Second, this approach leaves the nodes on one side of the partition idle, which reduces system performance and availability.

5.5 Summary

Table 9 summarizes the shortcomings of the current fault tolerance techniques, none of which are adequate for modern cloud systems. All current techniques severely affect system availability, as they unnecessarily lose a significant number of nodes. Failure verification and neutralizing partitioned nodes are used to fortify *specific* mechanisms, rather than providing *system-wide* fault tolerance. Using mechanism-specific fault tolerance techniques requires the independent fortification of all system mechanisms and the analysis of the interactions between various mechanisms. This approach complicates system design, fault analysis, and debugging. An example of a system that uses multiple mechanism-specific techniques to tolerate partial partitions is Elasticsearch, which uses checking neighbors' view, failure verification [61], and neutralizing partitioned nodes [64] in different mechanisms. However, Elasticsearch has the highest number of reported failures due to partial partitions (Table 1).

Detecting the surviving clique and checking neighbors' views can be used to build a *system-wide* fault tolerance technique. However, as Table 9 shows, these techniques lead to a complete system shutdown or significant loss of system capacity. This realization motivated us to build Nifty (Section 6), a system-wide fault tolerance technique that overcomes the aforementioned shortcomings.

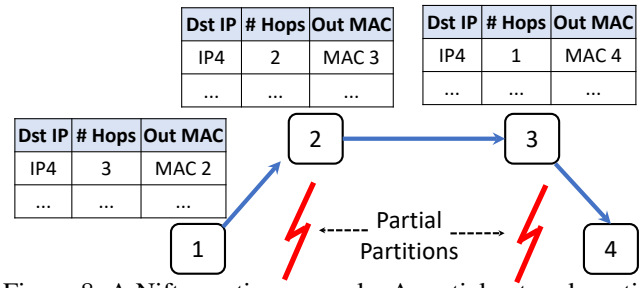


Figure 8: A Nifty routing example. A partial network partition isolates node 1 from nodes 3 and 4, and another partial partition isolates node 4 from nodes 1 and 2. Communication between 1 and 4 is routed through nodes 2 and 3.

6 Nifty Design

To overcome the limitations of current fault tolerance techniques, we design a simple, transparent network-partitioning fault-tolerant communication layer (Nifty).

Nifty follows a peer-to-peer design in which every node in the cluster runs a Nifty process. These processes collaborate in monitoring cluster connectivity. When Nifty detects a partial partition, it reroutes the traffic around the partition through intermediate nodes (i.e., bridge nodes). For instance, in Figure 8, if two partial partitions isolate node 1 from node 4, Nifty reroutes packets exchanged between nodes 1 and 4 through nodes 2 and 3.

Although Nifty keeps the cluster connected, it may increase the load on the bridge nodes, leading to a lower system performance. System designers who use Nifty may optimize the data or process placement or employ a flow-control mechanism to reduce the load on bridge nodes. To facilitate system-specific optimization, Nifty provides an API to identify bridge nodes.

Connectivity monitoring. Each Nifty process uses heart beating to monitor its connectivity with all other Nifty processes. Each Nifty process maintains a distance vector that includes the distance, in number of hops, to every node in the cluster. If a Nifty process misses three heartbeats from another Nifty process, it assumes that the communication with that process is broken and updates its distance vector. To detect when the communication between nodes recovers, Nifty processes continue to send heartbeats to disconnected nodes.

Recovery. Each Nifty process sends its distance vector (piggybacked on heartbeat messages) to all other nodes. Every Nifty process then uses these vectors to build and maintain a routing table.

When a Nifty process detects a change in the cluster (e.g., a node becomes unreachable or reachable), it initiates the route discovery procedure to find new routes. In our prototype, we use the classical Bellman–Ford distance-vector protocol [65, 66]. We use hop count as the link weight. By hop, we mean a hop between end nodes. Using hop count naturally favors direct connections, when they exist, over rerouting through intermediate nodes.

An entry in the routing table has a destination IP address, hop count, and output MAC address. If a packet is received with a destination IP address that matches an entry in the routing table, Nifty will change the destination MAC address of the packet to equal the output MAC address found in the routing table, then send the packet out.

Route deployment. Nifty uses OpenFlow [67] and Open vSwitch [68] to deploy the new routes. For instance, to reroute packets sent from node 1 to node 4 through nodes 2 and 3 in Figure 8, the Nifty process on node 1 installs rules on its local Open vSwitch to change the destination MAC address of any packet destined to node 4 to the MAC address of node 2. Whenever node 2 receives a packet with node 4 IP address as its destination, it changes the destination MAC address to node 3 MAC address and sends the packet out. Finally, when node 3 receives a packet with node 4 IP address, it changes the MAC address to node 4 MAC and sends the packet out.

Node classification. A system using Nifty can be optimized to reduce the amount of data forwarded through bridge nodes. The approach to do so is system-specific and may entail relocating processes in a cluster, dropping client requests, or reducing query result quality [7].

To facilitate the implementation of these mechanisms, Nifty identifies which nodes are on the same side of the network partition and which nodes serve as bridge nodes. It then provides this node classification to the system running atop of it. Section 7.3 demonstrates how this information can facilitate optimizing process placement in a VoltDB cluster.

7 Evaluation

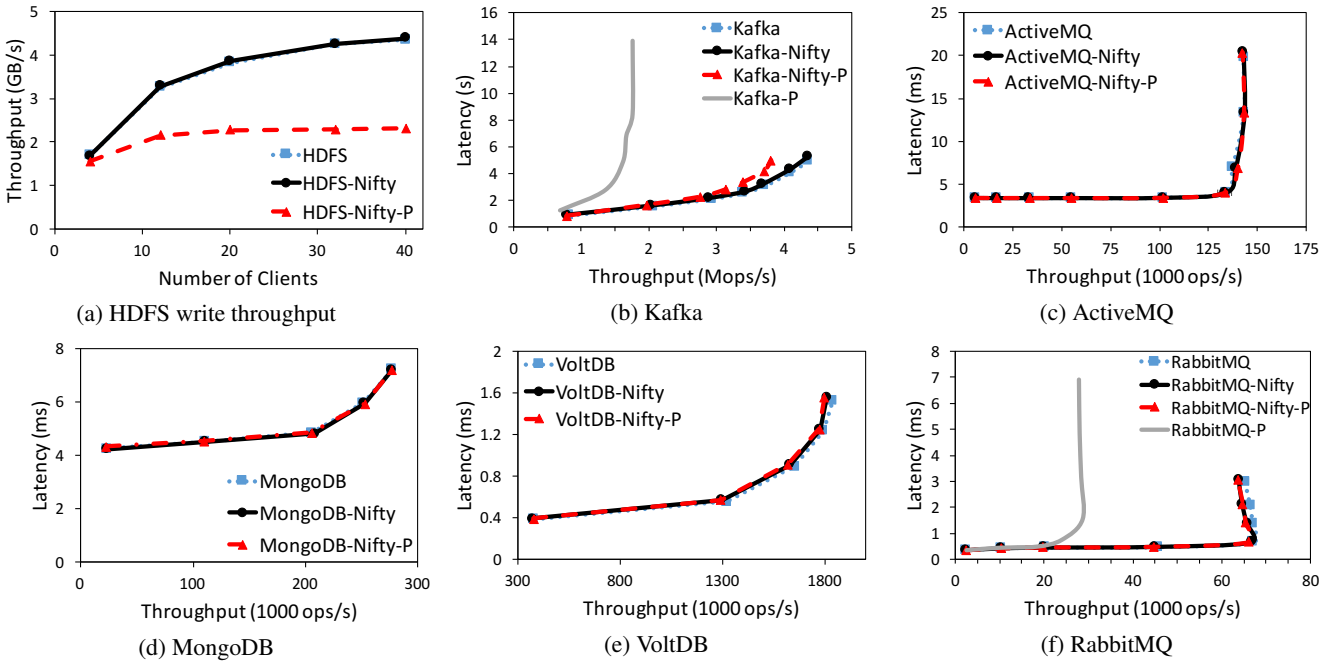
Our evaluation answers three questions. How much overhead does Nifty impose when there are no network partitions? What is a system's performance with Nifty under a network partition? What is the utility of Nifty's classification API?

Testbed. We conduct our experiments using 40 nodes at the Cloudlab Utah cluster. Each node has an Intel Xeon E5 10-core CPU, 64 GB of RAM, and a Mellanox ConnectX-4 25 Gbps NIC. To inject a network partition fault, we modify the Open vSwitch rules on the nodes to drop packets between the affected nodes. In all our experiments, we report the average for 30 runs. We note that the standard deviation in all our experiments is lower than 5%.

7.1 Overhead Evaluation

To evaluate Nifty's overhead, we measure its impact on the performance of a synthetic benchmark using iperf [69] and six data-centric systems (i.e., storage, database, and messaging systems). The iperf experiment uses a 100-node cluster to measure Nifty's impact on larger clusters. The systems we selected are:

- HDFS: We deploy HDFS (v3.3.0) on six nodes (one name node and five data nodes) and with a replication



(a) HDFS write throughput (b) Kafka (c) ActiveMQ

(d) MongoDB (e) VoltDB (f) RabbitMQ

Figure 9: Nifty's overhead. The average throughput for HDFS (a) and the average throughput vs. average latency for the rest of the systems. (-P) denotes the results with a partial partition.

level of three. To avoid disk access, we configure data nodes to use tmpfs. We use the HDFS standard benchmark (TestDFSIO). The benchmark reads and writes 1 GB files.

- Kafka: We deploy Kafka (v2.6.0) on five nodes. We distribute the queues (aka, topics) among nodes to balance the load. Each message is replicated on three nodes. We use Kafka's benchmarking tool to generate load on the system. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each queue has one consumer.

- ActiveMQ: We deploy ActiveMQ Artemis (v2.15.0) on five nodes with each queue being replicated on two nodes. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each queue has one consumer.

- MongoDB: We deploy MongoDB (v4.4.1) on six nodes (one config server and five mongod nodes) and with a replication level of three. We discuss our results with the Yahoo benchmark workload B (95% reads and 5% writes) with a uniform distribution [70]. We use 10 million records. The rest of the Yahoo benchmark workloads shows similar results.

- VoltDB: We deploy VoltDB (v9.0) on nine nodes, with data sharding enabled and a replication level of three. We use the Yahoo benchmark and the TCP-C benchmark. Figure 9.e shows the throughput-latency curve under Yahoo benchmark workload B (95% reads and 5% writes) with a uniform distribution. The results using the TPC-C benchmark and the Yahoo benchmark workloads A and C with uniform and skewed loads show similar low overhead.

- RabbitMQ: We deploy RabbitMQ (v3.8.2) on three nodes. We use the mirrored mode in which each queue has a leader replica and two backup replicas. We distribute the queue masters among brokers to distribute the load. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each consumer reads messages from a dedicated queue.

Results. We compare the throughput and average latency of each system with and without Nifty when there is no partial network partition. We evaluate Nifty with a partial partition in Section 7.2.

Figure 9 shows the write throughput of HDFS (Figure 9.a) and the throughput-latency curve for Kafka (Figure 9.b), ActiveMQ (Figure 9.c), MongoDB (Figure 9.d), VoltDB (Figure 9.e), and RabbitMQ (Figure 9.f). The results show that Nifty does not add noticeable overhead; for all systems, the curves almost completely overlap. This is because Nifty processes exchange a negligible number of packets. Each Nifty process sends a single UDP heartbeat packet every 200 ms to other nodes in the system. Consequently, in the largest deployment of nine nodes, each node sends only 40 packets every second.

Scalability evaluation. Nifty uses all-to-all heart beating to monitor a cluster's connectivity. Consequently, Nifty's overhead increases with the cluster size. To measure Nifty's scal-

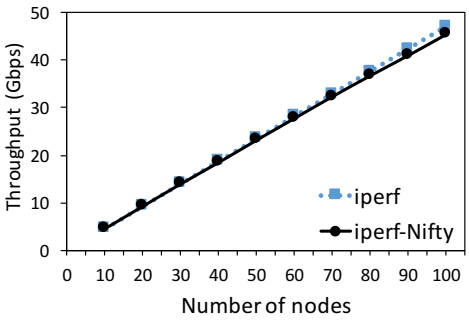


Figure 10: Scalability evaluation. Average throughput while increasing the number of nodes.

ability, we evaluate its overhead on a 100-node CloudLab Utah cluster. For this experiment, we limit the throughput of each node to 1 Gbps, as CloudLab can not support a full 10-Gbps connectivity between the 100 nodes we managed to book. To generate network intensive load, we use iperf [69]. Half of the nodes run an iperf server, and the other half run an iperf client. Each client communicates with a single server. Figure 10 shows the aggregate throughput of the iperf servers when deployed with and without Nifty. The figure shows that Nifty's overhead is negligible. When using 100 nodes, Nifty degrades the aggregate throughput by only 3.5%. Nevertheless, this monitoring approach will not scale to clusters with thousands of nodes. We are currently exploring the design of a fault tolerance technique that can scale to larger clusters.

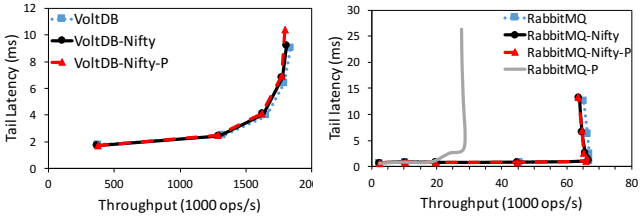
7.2 Handling Partial Partitions

To demonstrate the effectiveness of the proposed approach, we evaluate Nifty's performance with the six aforementioned systems under a partial partition fault. We note that RabbitMQ and VoltDB implemented two different techniques for tolerating partial partitions (Section 5).

Partial partition setup. We use the same deployment of the six aforementioned systems. Each system is deployed on an odd number of replicas. We introduce a partial partition that leaves one node as a bridge node and puts an equal number of nodes on each side of the partition. Client nodes are not affected by the partition. We partition the cluster this way to create maximum pressure on the bridge node.

Figure 9 shows the system performance when the cluster suffers from the partial partition. We notice that all the six systems are severely effected by the partial partition. HDFS, ActiveMQ, MongoDB, and VoltDB suffer a complete cluster pause or shutdown when deployed without Nifty. The VoltDB cluster shuts down because, after detecting the surviving clique, the system misses at least one shard. This confirms our analysis in Section 5.1.

RabbitMQ uses the checking neighbor's views fault tolerance approach. In our deployment, each queue is mirrored on a backup replica. Due to the strong consistency requirement, we configure RabbitMQ to pause in case of partial partition.



(a) VoltDB tail latency. (b) RabbitMQ tail latency.

Figure 11: Tail latency evaluation. Average throughput vs. 99th percentile of latency.

We deploy RabbitMQ on three nodes. Unfortunately, we could not use a larger RabbitMQ cluster because partial partitions often lead to the pause of the entire RabbitMQ cluster when Nifty is not used (Figure 4). Even with three nodes, partial partitions sometimes lead to pausing two out of three nodes. We discard those results and only include results in which one node pauses. Consequently, our results show the best possible performance of RabbitMQ under partial partitions. Pausing a broker in RabbitMQ leads to more than 50% reduction in throughput (RabbitMQ-P in (Figure 9.f)).

Kafka uses Zookeeper to monitor a cluster nodes. If a partial partition isolates a queue leader from the majority of replicas while Zookeeper runs on a bridge node, Zookeeper will not select a new leader and the entire cluster pauses (Finding 1 in Section 4). To mitigate this, we made sure that Zookeeper falls on one side of the partition. In this case, all the nodes on the other side of the partition that cannot reach Zookeeper are removed from the cluster. In our experiment, the partial partition causes two nodes to pause, which leads to almost a 50% reduction in system throughput (Figure 9.b).

Figure 9 shows that Nifty effectively masks the partial partition, so none of the nodes shut down or pause. Figure 9.a shows the write operation throughput for HDFS. With a replication level of three, each file has replicas on both sides of a partial partition. Consequently, for every 1 GB of data written, 1 or 2 GB of data are rerouted through the bridge node. This reduces the system throughput by up to 45%. We note that having a partial partition result in a performance degradation is better than a complete system unavailability when HDFS is deployed without Nifty. For the rest of the systems, during the partial partition, almost 50% of client requests and responses are rerouted through the bridge node. Even so, the system throughput only decreases by 2-6.7% and latency only increases by 3-7.8%. This shows that Nifty can effectively mask partial partitions and is able to utilize remaining connections to reduce the performance impact.

Figure 11 shows the tail latency for VoltDB and RabbitMQ for the same experiments presented in Figure 9. The figure shows the average throughput and the 99th percentile of latency while increasing the load on the system. The figure shows that Nifty increases the 99th percentile latency by up to 6.8% without a partial partition and by 15% under a partial partition failure.

7.3 Classification API Utility

In this section, we demonstrate the utility of Nifty's classification API. In VoltDB, a single server (aka, multi-data-partition initiator or MPI) processes all multi-shard operations. The MPI divides a multi-shard query (e.g., a join) to sub-queries, such that each sub-query targets a single shard. The MPI forwards each sub-query to its shard leader, gathers the intermediate results, performs final query processing, and sends the result to the client.

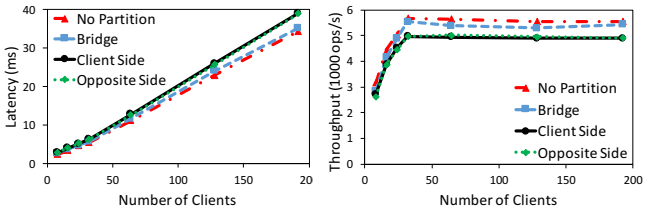
When deploying VoltDB atop Nifty, if the MPI node is on one side of the partition, a potentially significant volume of intermediate data passes through the bridge node. In our setup, when the MPI is on one side of the partition, 50% of the intermediate results are rerouted through the bridge node. This increases operation latency and the load on bridge nodes.

To improve the performance of multi-shard operations, the MPI process can be migrated to a bridge node. This effectively eliminates the need to reroute any traffic for multi-shard queries. We modify VoltDB to use Nifty's API to identify bridge nodes and migrate the MPI to a bridge node.

To evaluate this optimization's effectiveness, we evaluate the effect of the MPI's location on system performance. We restrict clients to contacting VoltDB nodes on one side of the partition and compare the system performance of three MPI placements: on clients side of the partition (client side in Figure 12), on the bridge node (bridge), and on the side opposite to the clients (opposite side). Bridge placement represents our optimization.

Setup and Workload. We use the same VoltDB configuration and partial partition setup detailed in the previouse sections. Unfortunately, VoltDB has limited support for join queries, so it cannot run standard benchmarks such as TPC-H [71]. In our experiments, we use a simple synthetic benchmark that joins two tables. The benchmark has two sharded tables of 20 fields each. Each field is 50 bytes, leading to approximately 1 KB rows. To use multiple shards, clients issue a range query that joins the two tables on the primary key. The client issues a query with a range that includes four primary keys. Consequently, the query result size is limited to four rows, with a total size of almost 8 KB. We populate the database with 20 GB of data before running the experiments. We report the average and standard deviation for 30 runs.

Results. Figure 12 shows the system throughput (a) and the average latency (b) for the three possible MPI placements. During a partial partition fault, placing the MPI on a bridge node decreases the latency by up to 11% and improves throughput by 11% compared to client and opposite side placements. Placing the MPI on a bridge node reduces the number of hops the join query must make before the MPI accumulates all the results and sends the query reply. Furthermore, bridge placement achieves throughput and latency within 4% of VoltDB's performance when there is no partition ("no partition" in Figure 12).



(a) Latency (b) Throughput

Figure 12: The impact of MPI placement on VoltDB's performance. Figure shows the average latency (a) and average throughput (b). Standard deviation was less than 2%.

We measure the amount of data forwarded through the bridge nodes for each one of those configurations; placing the MPI on the bridge node imposes the least overhead. When using 128 clients, 72 MB, 5 GB, and 6.5 GB of data are forwarded through the bridge node when the MPI is placed on the bridge, client side, and opposite side, respectively. The opposite side rerouts more data than the client side placement, as the client request and the result are also rerouted through the bridge node.

8 Related Work

To the best of our knowledge, this is the first study to focus on partial network partitioning, characterize its failures, dissect modern fault tolerance techniques, and explore the design of a generic fault tolerance technique for this type of fault.

A number of previous efforts analyzed failures in distributed systems, including characterizing specific component failures [5, 6, 72, 73, 74, 75] and characterizing failures in a specific domain such as HPC [76, 77, 78], IaaS clouds [79], data-mining services [80], hosting services [8, 81], data-intensive systems [82, 83, 84], and cloud systems [85]. Our work complements these efforts by focusing on failures triggered by partial network partitions.

In our previous work [12], we studied 136 network partitioning failures focusing on complete partitions. This previous work identified partial partitions, presented examples of how they can lead to system failures, and presented NEAT, a testing tool that can inject complete and partial network partitioning faults. We use NEAT to reproduce some of the reported failures. This paper presents an in-depth analysis of partial partition failures and fault tolerance techniques and proposes a novel fault-tolerant communication layer.

Comparing the characteristics of partial and complete partitions [12] shows that they have similar catastrophic impact and manifestation and reproducibility characteristics. Partial partitions seem easier to manifest. While all partial partition failures are triggered by a single-node partial partition and almost all of the failures are deterministic, 88% of the complete partitions manifest by isolating a single node and 80% of them are deterministic. Furthermore, we found twice as many failure reports reporting complete partitions than partial partitions.

Despite their similarity in causing catastrophic failures and being easy-to-manifest, partial and complete partitions are fundamentally different faults. Unlike complete partitions, a cluster suffering a partial partition is still connected but not all-to-all connected. Consequently, the CAP theorem bounds [13] do not apply to partial partitions. Furthermore, fault tolerance techniques for complete partitions cannot handle partial partitions or lead to pausing up to half of the cluster nodes. For instance, using majority vote to elect a leader is an effective mechanism to tolerate complete partitions. This approach alone is not effective in handling partial partitions, as there could be multiple completely connected subgroups with each connecting a majority of nodes. Section 5 shows how using only majority voting can lead to leader election thrashing and system unavailability.

Software-defined networking capabilities have been used to engineer traffic and optimize system operations, including offering network virtualization [86]; building network overlays [87]; performing network measurements [88, 89]; and implementing in-network firewalls [90], load balancers [91, 92], and key-value-based routing [93, 94]. Nifty is similar in spirit to these systems, as we use Open vSwitch capabilities to implement an overlay to mask partial partitions.

9 Concluding Remarks

Our work sheds light on a peculiar type of infrastructure fault and highlights the need for further research to understand such faults and explore techniques to improve systems' resiliency.

This is the first work to focus on partial network partitioning fault and present an in-depth analysis of system failures triggered by this fault. We identify characteristics that can facilitate better test design. Our findings highlight that focused design reviews can identify vulnerabilities early in the design process. We dissect the implementation of eight popular systems and study their fault tolerance techniques. In doing so, we identify four main approaches for tolerating partial partitions. Unfortunately, all implemented fault tolerance techniques have severe shortcomings.

We, therefore, build Nifty to overcome the limitations of modern fault tolerance techniques. Nifty is a simple, transparent communication layer that reroutes packets around partial partitions. We note that modern systems already incorporate a membership and connectivity monitoring. We show that extending the current implementations with a detour mechanism is an effective and low overhead fault tolerance technique to partial partitions. The source code for Nifty is available at https://github.com/UWASL/NIFTY

Acknowledgment

We thank the anonymous reviewers, our shepherd, Jason Flinn, Omid Abari, Ali Mashtizadeh, and Khuzaima Daudjee for their insightful feedback. We thank the artifact evaluation committee members for their effort in evaluating Nifty and for their feedback. We thank Joslin Goh for her feedback on our probabilistic analysis of VoltDB's failure probability. This research was supported by an NSERC Discovery grant, Canada Foundation for Innovation (CFI) grant, and a Waterloo-Huawei Joint Innovation lab grant.

References

[1] Daniel Turner, Kirill Levchenko, Jeffrey C Mogul, Stefan Savage, Alex C Snoeren, Daniel Turner, Kirill Levchenko, Jeffrey C Mogul, Stefan Savage, and Alex C Snoeren. On failure in managed enterprise networks. *HP Labs HPL-2012-101*, 2012.

[2] Data center: Load balancing data center, solutions reference nework design. Technical report, Cisco Systems, Inc., 2004.

[3] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72. ACM, 2016.

[4] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[5] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review*, 41(4):350–361, 2011.

[6] Daniel Turner, Kirill Levchenko, Alex C Snoeren, and Stefan Savage. California fault lines: understanding the causes and impact of network failures. *ACM SIGCOMM Computer Communication Review*, 41(4):315–326, 2011.

[7] Eric A Brewer. Lessons from giant-scale services. *IEEE Internet computing*, 5(4):46–55, 2001.

[8] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *USENIX symposium on internet technologies and systems*, volume 67. Seattle, WA, 2003.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[10] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[12] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, 2018.

[13] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[15] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[16] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, volume 95, pages 172–182, 1995.

[17] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[18] Rabbitmq message broker. https://www.rabbitmq.com. Accessed: Apr. 2020.

[19] Voltdb in-memory database platform. https://www.voltdb.com/. Accessed: Apr. 2020.

[20] The ceph object store. https://ceph.io/. Accessed: Apr. 2020.

[21] Robin J. Wilson. *Introduction to Graph Theory*. Prentice Hall/Pearson, New York, 2010.

[22] bnx2 cards intermittantly going offline. https://www.spinics.net/lists/netdev/msg152880.html. Accessed: Apr. 2020.

[23] Simon J Maple and Ian Robinson. Transaction recovery in a transaction processing computer system employing multiple transaction managers, October 20 2015. US Patent 9,165,025.

[24] Christian Maihofer. A survey of geocast routing protocols. *IEEE Communications Surveys & Tutorials*, 6(2):32–42, 2004.

[25] Matthew Milano and Andrew C Myers. Mixt: a language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 226–241. ACM, 2018.

[26] Observability in paxos clusters. https://davecturner.github.io/2017/08/18/observability-in-paxos.html. Accessed: Apr. 2020.

[27] Partial network partitions and obstacles to innovation. https://rachelbythebay.com/w/2012/02/16/partition/. Accessed:Apr. 2020.

[28] Partial network partition and retries. https://github.com/elastic/elasticsearch/issues/6105. Accessed: Apr. 2020.

[29] Healthchecking is not transitive. https://www.robustperception.io/healthchecking-is-not-transitive. Accessed: Apr. 2020.

[30] Cluster broken after switches upgrade. https://github.com/elastic/elasticsearch/issues/9495. Accessed: Apr. 2020.

[31] Using map output fetch failures to blacklist nodes is problematic. https://issues.apache.org/jira/browse/MAPREDUCE-1800. Accessed: Apr. 2020.

[32] Elasticsearch: Distributed search & analytics. https://www.elastic.co/products/elasticsearch. Accessed: Apr. 2020.

[33] Mongodb: The database for modern applications. https://www.mongodb.com/. Accessed: Apr. 2020.

[34] The apache hadoop project. http://hadoop.apache.org/. Accessed: Apr. 2020.

[35] Apache hbase. https://hbase.apache.org/. Accessed: Apr. 2020.

[36] Apache mesos. http://mesos.apache.org/. Accessed: Apr. 2020.

[37] Moosefs: Distributed file system. https://moosefs.com/. Accessed: Apr. 2020.

[38] Kafka: A distributed streaming platform. https://kafka.apache.org/. Accessed: Apr. 2020.

[39] Activemq: Flexible & powerful open source multi-protocol messaging. http://activemq.apache.org/. Accessed: Apr. 2020.

[40] Dkron: A distributed cron service. https://dkron.io/. Accessed: Apr. 2020.

[41] Robert V. Hogg, Elliot Tanis, and Dale Zimmerman. *Probability and Statistical Inference*. Pearson, 9 edition, 2013.

[42] Possible data loss when rs goes into gc pause while rolling hlog. https://issues.apache.org/jira/browse/HBASE-2312. Accessed: Apr. 2020.

[43] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[44] Activemq cluster blocks indefinitely in the presence of partial network partition. https://issues.apache.org/jira/browse/AMQ-7064. Accessed: Apr. 2020.

[45] Arbiters in pv1 should vote no in elections if they can see a healthy primary of equal or greater priority to the candidate. https://jira.mongodb.org/browse/SERVER-27125. Accessed: Apr. 2020.

[46] Partial network partition and retries. https://github.com/elastic/elasticsearch/issues/6105. Accessed: Apr. 2020.

[47] minimum_master_nodes does not prevent split-brain if splits are intersecting. https://github.com/elastic/elasticsearch/issues/2488. Accessed:Apr. 2020.

[48] Asymmetrical network partition can cause the election of two primary nodes. https://jira.mongodb.org/browse/SERVER-9730. Accessed: Apr. 2020.

[49] Mirrored queue crash with out of sync acks. https://github.com/rabbitmq/rabbitmq-server/issues/749. Accessed: Apr. 2020.

[50] A network partition can cause in flight documents to be lost. https://github.com/elastic/elasticsearch/issues/7572. Accessed: Apr. 2020.

[51] Hazelcast: the leading in-memory data grid. https://hazelcast.com/. Accessed: Apr. 2020.

[52] Redis: in-memory data structure store. https://redis.io/. Accessed: Apr. 2020.

[53] A. Herr. Veritas cluster server 6.2 I/O fencing deployment considerations. Technical report, Veritas Technologies, 2016.

[54] Balancer can cause cascading mongod failures during network partitions. https://jira.mongodb.org/browse/SERVER-19550. Accessed: Apr. 2020.

[55] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[56] Logcabin. https://github.com/logcabin/logcabin. Accessed: Apr. 2020.

[57] How does voltdb handle partial network partitions? https://www.voltdb.com/resources/transaction-consistency-faq#net. Accessed: Apr. 2020.

[58] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[59] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Understanding partial network partitioning. Technical Report WASL-TR-2020-02, Waterloo Advanced Systems Lab, University of Waterloo, October 2020.

[60] Partial network partitioning leads to cluster unavailability. https://github.com/elastic/elasticsearch/issues/43183. Accessed: Apr. 2020.

[61] Faulty recovery caused by partial network partitions. https://github.com/elastic/elasticsearch/pull/8720. Accessed: Apr. 2020.

[62] Mapreduce ticket 4832. https://issues.apache.org/jira/browse/MAPREDUCE-4832. Accessed: Apr. 2020.

[63] Designing highly available mesos frameworks. http://mesos.apache.org/documentation/latest/high-availability-framework-guide/. Accessed: Apr. 2020.

[64] Wait on shard failures. https://github.com/elastic/elasticsearch/issues/14252. Accessed: Apr. 2020.

[65] Deep Medhi and Karthik Ramasamy. *Network routing: algorithms, protocols, and architectures*. Morgan Kaufmann, 2017.

[66] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. *Data networks*, volume 2. Prentice-Hall International New Jersey, 1992.

[67] Openflow switch specification, version 1.5.1 (onf ts-025). Open Networking Foundation, 2015.

[68] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design

and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[69] iperf: The ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/. Accessed: Apr. 2020.

[70] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[71] TPC-H benchmark (decision support) standard specification. Transaction Processing Performance Council, December 2018. Revision 2.18.0.

[72] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204. ACM, 2010.

[73] Robert Birke, Ioana Giurgiu, Lydia Y Chen, Dorothea Wiesmann, and Ton Engbersen. Failure analysis of virtual and physical machines: patterns, causes and characteristics. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE, 2014.

[74] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. 2010.

[75] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage (TOS)*, 4(3):7, 2008.

[76] Nosayba El-Sayed and Bianca Schroeder. Reading between the lines of failure logs: Understanding how hpc systems fail. In *2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 1–12. IEEE, 2013.

[77] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. Bluegene/l failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 425–434. IEEE, 2006.

[78] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, 7(4):337–350, 2009.

[79] Theophilus Benson, Sambit Sahu, Aditya Akella, and Anees Shaikh. A first look at problems in the cloud. *HotCloud*, 10:15, 2010.

[80] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. An empirical study on quality issues of production big data platform. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 17–26. IEEE Press, 2015.

[81] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 1–16. ACM, 2016.

[82] Ariel Rabkin and Randy Howard Katz. How hadoop clusters break. *IEEE software*, 30(4):88–94, 2012.

[83] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[84] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 963–972. IEEE Press, 2013.

[85] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, 2014.

[86] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, 2018.

[87] Piyush Raman Srivastava and Saket Saurav. Networking agent for overlay l2 routing and overlay to underlay external networks l3 routing using openflow and open vswitch. In *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 291–296. IEEE, 2015.

[88] An Wang, Yang Guo, Songqing Chen, Fang Hao, TV Lakshman, Doug Montgomery, and Kotikalapudi Sriram. vprom: Vswitch enhanced programmable measurement in sdn. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2017.

[89] Zili Zha, An Wang, Yang Guo, Doug Montgomery, and Songqing Chen. Instrumenting open vswitch with monitoring capabilities: designs and challenges. In *Proceedings of the Symposium on SDN Research*, page 16. ACM, 2018.

[90] Pakapol Krongbaramee and Yuthapong Somchit. Implementation of sdn stateful firewall on data plane using open vswitch. In *2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–5. IEEE, 2018.

[91] Anat Bremler-Barr, David Hay, Idan Moyal, and Liron Schiff. Load balancing memcached traffic using software defined networking. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9. IEEE, 2017.

[92] Alex FR Trajano and Marcial P Fernandez. Two-phase load balancing of in-memory key-value storages through nfv and sdn. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 409–414. IEEE, 2015.

[93] I. Kettaneh, A. Alquraan, H. Takruri, S. Yang, A. S. Dusseau, R. Arpaci-Dusseau, and S. Al-Kiswany. The network-integrated storage system. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[94] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, cheap and in control with switchkv. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, 2016.



PACEMAKER
Avoiding HeART attacks in storage clusters with disk-adaptive redundancy

Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang,
K. V. Rashmi, Gregory R. Ganger
Carnegie Mellon University

Abstract

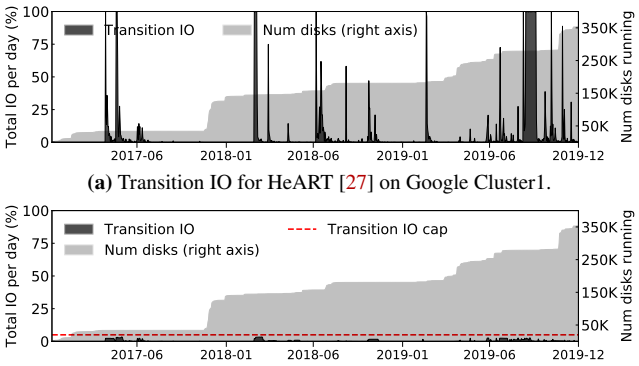
Data redundancy provides resilience in large-scale storage clusters, but imposes significant cost overhead. Substantial space-savings can be realized by tuning redundancy schemes to observed disk failure rates. However, prior design proposals for such tuning are unusable in real-world clusters, because the IO load of transitions between schemes overwhelms the storage infrastructure (termed *transition overload*).

This paper analyzes traces for millions of disks from production systems at Google, NetApp, and Backblaze to expose and understand transition overload as a roadblock to disk-adaptive redundancy: transition IO under existing approaches can consume 100% cluster IO continuously for several weeks. Building on the insights drawn, we present PACEMAKER, a low-overhead disk-adaptive redundancy orchestrator. PACEMAKER mitigates transition overload by (1) proactively organizing data layouts to make future transitions efficient, and (2) initiating transitions proactively in a manner that avoids urgency while not compromising on space-savings. Evaluation of PACEMAKER with traces from four large (110K–450K disks) production clusters show that the transition IO requirement decreases to never needing more than 5% cluster IO bandwidth (0.2–0.4% on average). PACEMAKER achieves this while providing overall space-savings of 14–20% and never leaving data under-protected. We also describe and experiment with an integration of PACEMAKER into HDFS.

1 Introduction

Distributed storage systems use data redundancy to protect data in the face of disk failures [13, 15, 56]. While it provides resilience, redundancy imposes significant cost overhead. Most large-scale systems today erasure code most of the data stored, instead of replicating, which helps to reduce the space overhead well below 100% [13, 24, 44, 48, 62, 67]. Despite this, space overhead remains a key concern in large-scale systems since it directly translates to an increase in the number of disks and the associated increase in capital, operating and energy costs [13, 24, 44, 48].

Storage clusters are made up of disks from a mix of makes/models acquired over time, and different makes/models have highly varying failure rates [27, 32, 41]. Despite that, storage clusters employ a "one-size-fits-all-disks" approach to choosing redundancy levels, without considering failure rate differences among disks. Hence, space overhead is often inflated by overly conservative redundancy levels, chosen to ensure sufficient protection for the most failure-prone disks in the cluster. Although tempting, the overhead cannot be removed by using very "wide" codes (which can provide high reliability with low storage overhead) for all data, due to the prohibitive reconstruction cost induced by the most failure-prone disks (more details in § 2). An exciting alternative is to dynamically adapt redundancy choices to observed failure rates (AFRs)[1] for different disks, which recent proposals suggest could substantially reduce the space overhead [27].



(a) Transition IO for HeART [27] on Google Cluster1.

(b) Transition IO for PACEMAKER on Google Cluster1.

Figure 1: Fraction of total cluster IO bandwidth needed to use disk-adaptive redundancy for a Google storage cluster's first three years. The state-of-the-art proposal [27] shown in (a) would require up to 100% of the cluster bandwidth for extended periods, whereas PACEMAKER shown in (b) always fits its IO under a cap (5%). The light gray region shows the disk count (right Y-axis) over time.

Adapting redundancy involves dynamic transitioning of redundancy schemes, because AFRs must be learned from observation of deployed disks and because AFRs change over time due to disk aging. Changing already encoded data from one redundancy scheme to another, for example from an erasure code with parameters k_1-of-n_1 to k_2-of-n_2 (where k-of-n denotes k data chunks and $n-k$ parity chunks; more details in § 2), can be exorbitantly IO intensive. Existing designs for disk-adaptive redundancy are rendered unusable by overwhelming bursts of urgent transition IO when applied to real-world storage clusters. Indeed, as illustrated in Fig. 1a, our analyses of production traces show extended periods of needing 100% of the cluster's IO bandwidth for transitions. We refer to this as the *transition overload* problem. At its core, transition overload occurs whenever an observed AFR increase for a subset of disks requires too much urgent transition IO in order to keep data safe. Existing designs for

[1] AFR describes the expected fraction of disks that experience failure in a typical year.

disk-adaptive redundancy perform redundancy transitions as a reaction to AFR changes. Since prior designs are reactive, for an increase in AFR, the data is already under-protected by the time the transition to increase redundancy is issued. And it will continue to be under-protected until that transition completes. For example, around 2019-09 in Fig. 1a, data was under-protected for over a month, even though the entire cluster's IO bandwidth was used solely for redundancy transitions. Simple rate-limiting to reduce urgent bursts of IO would only exacerbate this problem causing data-reliability goals to be violated for even longer.

To understand the causes of transition overload and inform solutions, we analyse multi-year deployment and failure logs for over 5.3 million disks from Google, NetApp and Backblaze. Two common transition overload patterns are observed. First, sometimes disks are added in tens or hundreds over time, which we call *trickle* deployments. A statistically confident AFR observation requires thousands of disks. Thus, by the time it is known that AFR for a specific make/model and age is too high for the redundancy used, the oldest thousands of that make/model will be past that age. At that point, all of those disks need immediate transition. Second, sometimes disks are added in batches of many thousands, which we call *step* deployments. Steps have sufficient disks for statistically confident AFR estimation. However, when a step reaches an age where the AFR is too high for the redundancy used, *all* disks of the step need immediate transition.

This paper introduces PACEMAKER, a new disk-adaptive redundancy orchestration system that exploits insights from the aforementioned analyses to eliminate the transition overload problem. PACEMAKER proactively organizes data layouts to enable efficient transitions for each deployment pattern, reducing total transition IO by over 90%. Indeed, by virtue of its reduced total transition IO, PACEMAKER can afford to use extra transitions to reap increased space-savings. PACEMAKER also proactively initiates anticipated transitions sufficiently in advance that the resulting transition IO can be rate-limited without placing data at risk. Fig. 1b provides a peek into the final result: PACEMAKER achieves disk-adaptive redundancy with substantially less total transition IO and never exceeds a specified transition IO cap (5% in the graph).

We evaluate PACEMAKER using logs containing all disk deployment, failure, and decommissioning events from four production storage clusters: three 160K–450K-disk Google clusters and a ≈110K-disk cluster used for the Backblaze Internet backup service [4]. On all four clusters, PACEMAKER provides disk-adaptive redundancy while using less than 0.4% of cluster IO bandwidth for transitions on average, and never exceeding the specified rate limit (e.g., 5%) on IO bandwidth. Yet, despite its proactive approach, PACEMAKER loses less than 3% of the space-savings as compared to to an idealized system with perfectly-timed and instant transitions. Specifically, PACEMAKER provides 14–20% average space-savings compared to a one-size-fits-all-disks approach, without ever failing to meet the target data reliability and with no transition overload. We note that this is substantial savings for large-scale systems, where even a single-digit space-savings is worth the engineering effort. For example, in aggregate, the four clusters would need ≈200K fewer disks.

We also implement PACEMAKER in HDFS, demonstrating that PACEMAKER's mechanisms fit into an existing cluster storage system with minimal changes. Complementing our longitudinal evaluation using traces from large scale clusters, we report measurements of redundancy transitions in PACEMAKER-enhanced HDFS via small-scale cluster experiments. Prototype of HDFS with Pacemaker is open-sourced and is available at `https://github.com/thesys-lab/pacemaker-hdfs.git`.

This paper makes five primary contributions. First, it demonstrates that transition overload is a roadblock that precludes use of previous disk-adaptive redundancy proposals. Second, it presents insights into the sources of transition overload from longitudinal analyses of deployment and failure logs for 5.3 million disks from three large organizations. Third, it describes PACEMAKER's novel techniques, designed based on insights drawn from these analyses, for safe disk-adaptive redundancy without transition overload. Fourth, it evaluates PACEMAKER's policies for four large real-world storage clusters, demonstrating their effectiveness for a range of deployment and disk failure patterns. Fifth, it describes integration of and experiments with PACEMAKER's techniques in HDFS, demonstrating their feasibility, functionality, and ease of integration into a cluster storage implementation.

2 Whither disk-adaptive redundancy

Cluster storage systems and data reliability. Modern storage clusters scale to huge capacities by combining up to hundreds of thousands of storage devices into a single storage system [15,56,63]. In general, there is a metadata service that tracks data locations (and other metadata) and a large number of storage servers that each have up to tens of disks. Data is partitioned into chunks that are spread among the storage servers/devices. Although hot/warm data is now often stored on Flash SSDs, cost considerations lead to the majority of data continuing to be stored on mechanical disks (HDDs) for the foreseeable future [6,7,54]. For the rest of the paper, any reference to a "device" or "disk" implies HDDs.

Disk failures are common and storage clusters use data redundancy to protect against irrecoverable data loss in the face of disk failures [4,15,24,41,43,44,48]. For hot data, often replication is used for performance benefits. But, for most bulk and colder data, cost considerations have led to the use of erasure coding schemes. Under a k-of-n coding scheme, each set of k data chunks are coupled with n-k "parity chunks" to form a "stripe". A k-of-n scheme provides tolerance to $(n-k)$ failures with a space overhead of $\frac{n}{k}$. Thus, erasure coding achieves substantially lower space overhead for tolerating a given number of failures. Schemes like 6-of-9 and 10-of-14

are commonly used in real-world deployments [13, 43, 44, 48]. Under erasure coding, additional work is involved in recovering from a device failure. To reconstruct a lost chunk, k remaining chunks from the stripe must be read.

The redundancy scheme selection problem. The reliability of data stored redundantly is often quantified as *mean-time-to-data-loss* (MTTDL) [17], which essentially captures the average time until more than the tolerated number of chunks are lost. MTTDL is calculated using the disks' AFR and its *mean-time-to-repair* (MTTR).

Large clusters are built over time, and hence usually consist of a mix of disks belonging to multiple makes/models depending on which options were most cost effective at each time. AFR values vary significantly between makes/models and disks of different ages [27, 32, 41, 50]. Since disks have different AFRs, computing MTTDL of a candidate redundancy scheme for a large-scale storage cluster is often difficult.

The MTTDL equations can still be used to guide decisions, as long as a sufficiently high AFR value is used. For example, if the highest AFR value possible for any deployed make/model at any age is used, the computed MTTDL will be a lower bound. So long as the lower bound on MTTDL meets the target MTTDL, the data is adequately reliable. Unfortunately, the range of possible AFR values in a large storage cluster is generally quite large (over an order of magnitude) [27, 32, 41, 52]. Since the overall average is closer to the lower end of the AFR range, the highest AFR value is a conservative over-estimate for most disks. The resulting MTTDLs are thus loose lower bounds, prompting decision-makers to use a one-size-fits-all scheme with excessive redundancy leading to wasted space.

Using wide schemes with large number of parities (e.g., 30-of-36) can achieve the desired MTTDL while keeping the storage overhead low enough to make disk-adaptive redundancy appear not worth the effort. But, while this might seem like a panacea, wide schemes in high-AFR regimes cause significant increase in failure reconstruction IO traffic. The failure reconstruction IO is derived by multiplying the AFR with the number of data chunks in each stripe. Thus, if either of these quantities are excessively high, or both are moderately high, it can lead to overwhelmingly high failure reconstruction IO. In addition, wide schemes also result in higher tail latencies for individual disk reconstructions because of having to read from many more disks. Combined, these reasons prevent use of wide schemes for all data all the time from being a viable solution for most systems.

Disk-adaptive redundancy. Since the problem arises from using a single AFR value, a promising alternative is to adapt redundancy for subsets of disks with similar AFRs. A recent proposal, heterogeneity-aware redundancy tuner (HeART) [27], suggests treating subsets of deployed disks with different AFR characteristics differently. Specifically, HeART adapts redundancy of each disk by observing its failure rate on the fly[2] depending on its make/model and its current age. It is well known that AFR of disks follow a "bathtub" shape with three distinct phases of life: AFR is high in "infancy" (1-3 months), low and stable during its "useful life" (3-5 years), and high during the "wearout" (a few months before decommissioning). HeART uses a default (one-size-fits-all) redundancy scheme for each new disk's infancy. It then dynamically changes the redundancy to a scheme adapted to the observed useful life AFR for that disk's make/model, and then dynamically changes back to the default scheme at the end of useful life. The per-make/model useful life redundancy schemes typically have much lower space overhead than the default scheme. This suggests the ability to maintain target MTTDL with many fewer disks (i.e., lower cost).

Although exciting, the design of HeART overlooks a crucial element: the IO cost associated with changing the redundancy schemes. Changing already encoded data under one erasure code to another can be exorbitantly IO intensive. Indeed, our evaluation of HeART on real-world storage cluster logs reveal extended periods where data safety is at risk and where 100% cluster IO bandwidth is consumed for scheme changes. We call this problem *transition overload*.

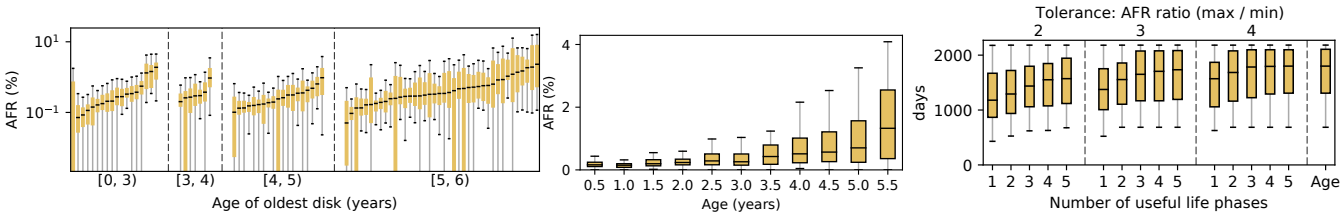
An enticing solution that might appear to mitigate transition overload is to adapt redundancy schemes only by removing parities in low-AFR regimes and adding parities in high-AFR regimes. While this solution eliminates transition IO when reducing the level of redundancy, it does only marginally better when redundancy needs to be increased, because new parity creation cannot avoid reading all data chunks from each stripe. What makes this worse is that transitions that increase redundancy are time-critical, since delaying them would miss the MTTDL target and leave the data under-protected. Moreover, addition / removal of a parity chunk massively changes the stripe's MTTDL compared to addition / removal of a data chunk. For example, a 6-of-9 MTTDL is 10000× higher than 6-of-8 MTTDL, but is only 1.5× higher than 7-of-10 MTTDL. AFR changes would almost never be large enough to safely remove a parity, given default schemes like 6-of-9, eliminating almost all potential benefits of disk-adaptive redundancy.

This paper analyzes disk deployment and failure data from large-scale production clusters to discover sources of *transition overload* and informs the design of a solution. It then describes and evaluates PACEMAKER, which realizes the dream of safe disk-adaptive redundancy without transition overload.

3 Longitudinal production trace analyses

This section presents an analysis of multi-year disk reliability logs and deployment characteristics of 5.3 million HDDs, covering over 60 makes/models from real-world environments. Key insights presented here shed light on the

[2] Although it may be tempting to use AFR values taken from manufacturer's specifications, several studies have shown that failure rates observed in practice often do not match those [41, 50, 52].



(a) Spread of make/model AFRs **(b)** AFR distribution over disk life **(c)** Approximate useful-life length

Figure 2: (a) AFR spread for over 50 makes/models from NetApp binned by the age of the oldest disk. Each box corresponds to a unique make/model, and at least 10000 disks of each make/model were observed (outlier AFR values omitted). (b) Distribution of AFR calculated over consecutive non-overlapping six-month periods for NetApp disks, showing the gradual rise of AFR with age (outliers omitted). (c) Approximation of useful life length for NetApp disks for 1-5 consecutive phases of useful life and three different tolerance levels.

sources of transition overload and challenges / opportunities for a robust disk-adaptive redundancy solution.

The data. Our largest dataset comes from NetApp and contains information about disks deployed in filers (file servers). Each filer reports the health of each disk periodically (typically once a fortnight) using their AutoSupport [29] system. We analyzed the data for a subset of their deployed disks, which included over 50 makes/models and over 4.3 million disks total. As observed in previous studies [27, 41, 50], we observe well over an order of magnitude difference between the highest and lowest useful-life AFRs (see Fig. 2a).

Our other datasets come from large storage clusters deployed at Google and the Backblaze Internet backup service. Although the basic disk characteristics (e.g., AFR heterogeneity and its behavior discussed below) are similar to the NetApp dataset, these datasets also capture the evolution and behavior in our target context (large-scale storage clusters), and thus are also used in the evaluation detailed in (§7). The particular Google clusters were selected based on their longitudinal data availability, but were not otherwise screened for favorability.

For each cluster, the multi-year log records (daily) all disk deployment, failure, and decommissioning events from birth of the cluster until the date of the log snapshot. Google Cluster1's disk population over three years included ≈350K disks of 7 makes/models. Google Cluster2's population over 2.5 years included ≈450K disks of 4 makes/models. Google Cluster3's population over 3 years included ≈160K disks of 3 makes/models. The Backblaze cluster's population since 2013 included ≈110K disks of 7 makes/models.

3.1 Causes of transition overload

Disk deployment patterns. We observe disk deployments occurring in two distinct patterns, which we label ***trickle*** and ***step***. Trickle-deployed disks are added to a cluster frequently (weekly or even daily) over time by the tens and hundreds. For example, the slow rise in disk count seen between 2018-01 and 2018-07 in Fig. 1 represents a series of trickle-deployments. In contrast, a step-deployment introduces many thousands of disks into the cluster "at once" (over a span of a few days), followed by potentially months of no new step-deployments. The sharp rises in disk count around 2017-12 and 2019-11 in Fig. 1 represent step-deployments.

A given cluster may be entirely trickle-deployed (like the Backblaze cluster), entirely step-deployed (like Google Cluster2), or a mix of the two (like Google Cluster1 and Cluster3). Disks of a step are typically of the same make/model.

Learning AFR curves online. Disk-adaptive redundancy involves learning the AFR curve for each make/model by observing failures among deployed disks of that make/model. Because AFR is a statistical measure, the larger the population of disks observed at a given age, the lower is the uncertainty in the calculated AFR at that age. We have found that a few thousand disks need to be observed to obtain sufficiently accurate AFR measurements.

Transition overload for trickle-deployed disks. Since trickle-deployed disks are deployed in tiny batches over time, several months can pass before the required number of disks of a new make/model are past any given age. Thus, by the time the required number of disks can be observed at the age that is eventually identified as having too-high an AFR and requiring increased redundancy, data on the older disks will have been left under-protected for months. And, the thousands of already-older disks need to be immediately transitioned to a stronger redundancy scheme, together with the newest disks to reach that age. This results in transition overload.

Transition overload for step-deployed disks. Assuming that they are of the same make/model, a batch of step-deployed disks will have the same age and AFR, and indeed represent a large enough population for confident learning of the AFR curve as they age. But, this means that all of those disks will reach AFR values together, as they age. So, when their AFR rises to the point where the redundancy must be increased to keep data safe, all of the disks must transition together to the new safer redundancy scheme. Worse, if they are the first disks of the given make/model deployed in the cluster, which is often true in the clusters studied, then the system adapting the redundancy will learn of the need only when the age in question is reached. At that point, all data stored on the entire batch of disks is unsafe and needs immediate transitioning. This results in transition overload.

3.2 Informing a solution

Analyzing the disk logs has exposed a number of observations that provide hope and guide the design of PACEMAKER. The AFR curves we observed deviate substantially from the canonical representation where infancy and wearout periods

are identically looking and have high AFR values, and AFR in useful life is flat and low throughout.

AFRs rise gradually over time with no clear wearout. AFR curves generally exhibit neither a flat useful life phase nor a sudden transition to so-called wearout. Rather, in general, it was observed that AFR curves rise gradually as a function of disk age. Fig. 2b shows the gradual rise in AFR over six month periods of disk lifetimes. Each box represents the AFR of disks whose age corresponds to the six-month period denoted along the X-axis. AFR curves for individual makes/models (e.g., Figs. 5b and 5d) are consistent with this aggregate illustration. Importantly, none of the over 60 makes/models from Google, Backblaze and NetApp displayed sudden onset of wearout.

Gradual increases in AFR, rather than sudden onset of wearout, suggests that one could anticipate a step-deployed batch of disks approaching an AFR threshold. This is one foundation on which PACEMAKER's proactive transitioning approach rests.

Useful life could have multiple phases. Given the gradual rise of AFRs, useful life can be decomposed into multiple, piece-wise constant phases. Fig. 2c shows an approximation of the length of useful life when multiple phases are considered. Each box in the figure represents the distribution over different make/models of the approximate length of useful life. Useful life is approximated by considering the longest period of time which can be decomposed into multiple consecutive phases (number of phases indicated by the bottom X-axis) such that the ratio between the maximum and minimum AFR in each phase is under a given tolerance level (indicated by the top X-axis). The last box indicates the distribution over make/models of the age of the oldest disk, which is an upper bound to the length of useful life. As shown by Fig. 2c, the length of useful life can be significantly extended (for all tolerance levels) by considering more than one phase. Furthermore, the data show that a small number of phases suffice in practice, as the approximate length of useful life changes by little when considering four or more phases.

Infancy often short-lived. Disks may go through (potentially) multiple rounds of so-called "burn-in" testing. The first tests may happen at the manufacturer's site. There may be additional burn-in tests done at the deployment site allowing most of the infant mortality to be captured before the disk is deployed in production. For the NetApp and Google disks, we see the AFR drop sharply and plateau by 20 days for most of the makes/models. In contrast, the Backblaze disks display a slightly longer and higher AFR during infancy, which can be directly attributed to their less aggressive on-site burn-in.

PACEMAKER's design is heavily influenced from these learnings, as will be explained in the next section.

4 Design goals

PACEMAKER is an IO efficient redundancy orchestrator for storage clusters that support disk-adaptive redundancy.

Term	*Definition*
Dgroup	Group of disks of the same make/model.
Transition	The act of changing the redundancy scheme.
RDn transition	Transition to a lower level of redundancy.
RUp transition	Transition to a higher level of redundancy.
peak-IO-cap	IO bandwidth cap for transitions.
Rgroup	Group of disks using the same redundancy with placement restricted to the group of disks.
Rgroup0	Rgroup using the default one-scheme-fits-all redundancy used in storage clusters today.
Unspecialized disks	Disks that are a part of Rgroup0.
Specialized disks	Disks that are not part of Rgroup0.
Canary disks	First few thousand disks of a trickle-deployed Dgroup used to learn AFR curve.
Tolerated-AFR	Max AFR for which redundancy scheme meets reliability constraint.
Threshold-AFR	The AFR threshold crossing which triggers an RUp transition for step-deployed disks.

Table 1: Definitions of PACEMAKER's terms.

Before going into the design goals for PACEMAKER, we first chronicle a disk's lifecycle, introducing the terminology that will be used in the rest of the paper (defined in Table 1).

Disk lifecycle under PACEMAKER. Throughout its life, each disk under PACEMAKER simultaneously belongs to a *Dgroup* and an *Rgroup*. There are as many Dgroups in a cluster as there are unique disk makes/models. Rgroups on the other hand are a function of redundancy schemes and placement restrictions. Each Rgroup has an associated redundancy scheme, and its data (encoded stripes) must reside completely within that Rgroup's disks. Multiple Rgroups can use the same redundancy scheme, but no stripe may span across Rgroups. The Dgroup of a disk never changes, but a disk may transition through multiple Rgroups during its lifetime. At the time of deployment (or "birth"), the disk belongs to *Rgroup0*, and is termed as an *unspecialized disk*. Disks in Rgroup0 use the default redundancy scheme, i.e. the conservative one-scheme-fits-all scheme used in storage clusters that do not have disk-adaptive redundancy. The redundancy scheme employed for a disk (and hence its Rgroup) changes via *transitions*. The first transition any disk undergoes is an *RDn transition*. A RDn transition changes the disk's Rgroup to one with lower redundancy, i.e. more optimized for space. Whenever the disk departs from Rgroup0, it is termed as a *specialized disk*. Disks depart from Rgroup0 at the end of their infancy. Since infancy is short-lived (§3.2), PACEMAKER only considers one RDn transition for each disk.

The first RDn transition occurs at the start of the disk's useful life, and marks the start of its specialization period. As explained in §3.2, a disk may experience multiple useful life phases. PACEMAKER performs a transition at the start of each useful life phase. After the first (and only) RDn transition, each subsequent transition is an *RUp transition*. An RUp transition changes the disk's Rgroup to one with higher redundancy, i.e. less optimized for space, but the disk is still considered a specialized disk unless the Rgroup that the disk is being RUp transitioned to is Rgroup0. The space-

savings (and thus cost-savings) associated with disk-adaptive redundancy are proportional to the fraction of life the disks remain specialized for.

Key decisions. To adapt redundancy throughout a disk's lifecycle as chronicled above, three key decisions related to transitions must be made

1. ***When should the disks transition?***
2. ***Which Rgroup should the disks transition to?***
3. ***How should the disks transition?***

Constraints. The above decisions need to be taken such that a set of constraints are met. An obvious constraint, central to any storage system, is that of data reliability. The *reliability constraint* mandates that all data must always meet a predefined target MTTDL. Another important constraint is the *failure reconstruction IO constraint*. This constraint bounds the IO spent on data reconstruction of failed disks, which as explained in §2 is proportional to AFR and scheme width. This is why wide schemes cannot be used for all disks all the time, but they can be used for low-AFR regimes of disk lifetimes (as discussed in §2).

Existing approaches to disk-adaptive redundancy make their decisions on the basis of only these constraints [27], but fail to consider the equally important *IO caused by redundancy transitions*. Ignoring this causes the transition overload problem, which proves to be a show-stopper for disk-adaptive redundancy systems. PACEMAKER treats transition IO as a first class citizen by taking it into account for each of its three key decisions. As such, PACEMAKER enforces carefully designed constraints on transition IO as well.

Designing IO constraints on transitions. Apart from serving foreground IO requests, a storage cluster performs numerous background tasks like scrubbing and load balancing [5,38,49]. Redundancy management is also a background task. In current storage clusters, redundancy management tasks predominantly consist of performing data redundancy (e.g. replicating or encoding data) and reconstructing data of failed or otherwise unavailable disks. Disk-adaptive redundancy systems add redundancy transitions to the list of IO-intensive background tasks.

There are two goals for background tasks: Goal 1: they are not too much work, and Goal 2: they interfere as little as possible with foreground IO. PACEMAKER applies two IO constraints on background transition tasks to achieve these goals: (1) *average-IO constraint* and (2) *peak-IO constraint*. The average-IO constraint achieves Goal 1 by allowing storage administrators to specify a cap on the fraction of the IO bandwidth of a disk that can be used for transitions over its lifetime. For example, if a disk can transition in 1 day using 100% of its IO bandwidth, then an average-IO constraint of 1% would mean that the disk will transition at most once every 100 days. The peak-IO constraint achieves Goal 2 by allowing storage administrators to specify the peak rate (defined as the *peak-IO-cap*) at which transitions can occur so as to limit their interference with foreground traffic. Continuing the previous example, if the peak-IO-cap is set at 5%, the disk that would have taken 1 day to transition at 100% IO bandwidth would now take at least 20 days. The average-IO constraint and the peak-IO-cap can be configured based on how busy the cluster is. For example, a cluster designed for data archival would have a lower foreground traffic, compared to a cluster designed for serving ads or recommendations. Thus, low-traffic clusters can set a higher peak-IO-cap resulting in faster transitions and potentially increased space-savings.

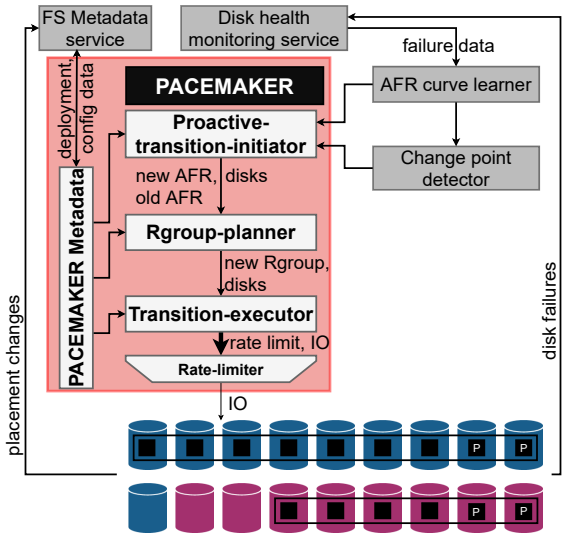


Figure 3: PACEMAKER architecture.

Design goals. The key design goals are to answer the three questions related to transitions such that the space-savings are maximized and the following constraints are met: (1) reliability constraint on all data all the time, (2) failure reconstruction IO constraint on all disks all the time, (3) peak-IO constraint on all disks all the time, and (4) average-IO constraint on all disks over time.

5 Design of PACEMAKER

Fig. 3 shows the high level architecture of PACEMAKER and how it interacts with some other components of a storage cluster. The three main components of PACEMAKER correspond to the three key decisions that the system makes as discussed in §4. The first main component of PACEMAKER is the ***proactive-transition-initiator*** (§5.1), which determines when to transition disks using the AFR curves and the disk deployment information. The information of the transitioning disks and their observed AFR is passed to the ***Rgroup-planner*** (§5.2), which chooses the Rgroup to which the disks should transition. The Rgroup-planner passes the information of the transitioning disks and the target Rgroup to the ***transition-executor*** (§5.3). The transition-executor addresses how to transition the disks to the planned Rgroup in the most IO-efficient way.

Additionally, PACEMAKER also maintains its own ***metadata*** and a simple ***rate-limiter***. PACEMAKER metadata interacts with all of PACEMAKER's components and also the

storage cluster's metadata service. It maintains various configuration settings of a PACEMAKER installation along with the disk deployment information that guides transition decisions. The rate-limiter rate-limits the IO load generated by any transition as per administrator specified limits. Other cluster components external-to-PACEMAKER that inform it are the *AFR curve learner* and the *change point detector*. As is evident from their names, these components learn the AFR curve[3] of each Dgroup and identify change points for redundancy transitions. The AFR curve learner receives failure data from the *disk health monitoring service*, which monitors the disk fleet and maintains their vitals.

5.1 Proactive-transition-initiator

Proactive-transition-initiator's role is to determine *when to transition the disks*. Below we explain PACEMAKER's methodology for making this decision for the two types of transitions (RDn and RUp) and the two types of deployments (step and trickle).

5.1.1 Deciding when to RDn a disk

Recall that a disk's first transition is an RDn transition. As soon as proactive-transition-initiator observes (in a statistically accurate manner) that the AFR has decreased sufficiently, and is stable, it performs an RDn transition from the default scheme (i.e., from Rgroup0) employed in infancy to a more space-efficient scheme. This is the only RDn transition in a disk's lifetime.

5.1.2 Deciding when to RUp a disk

RUp transitions are performed either when there are too few disks in any Rgroup such that data placement is heavily restricted (which we term *purging an Rgroup*), or when there is a rise in AFR such that the reliability constraint is (going to be) violated. Purging an Rgroup involves RUp transitioning all of its disks to an Rgroup with higher redundancy. This transition isn't an imminent threat to reliability, and therefore can be done in a relaxed manner without violating the reliability constraint as explained in §5.3.

However, most RUp transitions in a storage cluster are done in response to a rise in AFR. These are challenging with respect to meeting IO constraints due to the associated risk of violating the reliability constraints whenever the AFR rises beyond the AFR tolerated by the redundancy scheme (termed *tolerated-AFR*).

In order to be able to safely rate-limit the IO load due to RUp transitions, PACEMAKER takes a *proactive* approach. The key is in determining when to initiate a proactive RUp transition such that the transition can be completed before the AFR crosses the tolerated-AFR, while adhering to the IO and the reliability constraints without compromising much on space-savings. To do so, the proactive-transition-initiator assumes that its transitions will proceed as per the peak-IO constraint, which is ensured by the transition-executor. PACEMAKER's methodology for determining when to initiate a proactive RUp transition is tailored differently for trickle versus for step deployments, since they raise different challenges.

Trickle deployments. For trickle-deployed disks, PACEMAKER considers two category of disks: (1) first disks to be deployed from any particular trickle-deployed Dgroup, and (2) disks from that Dgroup that are deployed later.

PACEMAKER labels the first C deployed disks of a Dgroup as *canary* disks, where C is a configurable, high enough number of disks to yield statistically significant AFR observations. For example, based on our disk analyses, we observe that C in low thousands (e.g., 3000) is sufficient. The canary disks of any Dgroup are the first to undergo the various phases of life for that Dgroup, and these observations are used to learn the AFR curve for that Dgroup. The AFR value for the Dgroup at any particular age is not known (with statistical confidence) until all canary disks go past that age. Furthermore, due to the trickle nature of the deployment, the canary disks would themselves have been deployed over weeks if not months. Thus, AFR for the canary disks can be ascertained only in retrospect. PACEMAKER never changes the redundancy of the canary disks to avoid them from ever violating the reliability constraint. This does not significantly reduce space-savings, since C is expected to be small relative to the total number of disks of a Dgroup (usually in the tens of thousands).

The disks that are deployed later in any particular Dgroup are easier to handle, since the Dgroup's AFR curve would have been learned by observing the canaries. Thus, the date at which a disk among the later-deployed disks needs to RUp to meet the reliability constraints is known in advance by the proactive-transition-initiator, which it uses to issue proactive RUp transitions.

Step deployments. Recall that in a step deployment, most disks of a Dgroup may be deployed within a few days. So, canaries are not a good solution, as they would provide little-to-no advance warning about how the AFR curve's rises would affect most disks.

PACEMAKER's approach to handling step-deployments is based on two properties: (1) Step-deployments have a large number of disks deployed together, leading to a statistically accurate AFR estimation; (2) AFR curves based on a large set of disks tend to exhibit gradual, rather than sudden, AFR increases as the disk ages (§3.2). PACEMAKER leverages these two properties to employ a simple *early warning* methodology to predict a forthcoming need to RUp transition a step well in advance. Specifically, PACEMAKER sets a threshold, termed *threshold-AFR*, which is a (configurable) fraction of the tolerated-AFR of the current redundancy scheme employed. For step-deployments, when the observed AFR crosses the threshold-AFR, the proactive-transition-initiator initiates a proactive RUp transition.

5.2 Rgroup-planner

The Rgroup-planner's role is to determine *which Rgroup should disks transition to*. This involves making two *inter-*

[3] The AFR estimation methodology employed is detailed in [26].

dependent choices: (1) the redundancy scheme to transition into, (2) whether or not to create a new Rgroup.

Choice of the redundancy scheme. At a high level, the Rgroup-planner first uses a set of selection criteria to arrive at a set of viable schemes. It further narrows down the choices by filtering out the schemes that are not worth transitioning to when the transition IO and IO constraints are accounted for.

Selection criteria for viable schemes. Each viable redundancy scheme has to satisfy the following criteria in addition to the reliability constraint: each scheme (1) must satisfy the minimum number of simultaneous failures per stripe (i.e., $n-k$); (2) must not exceed the maximum allowed stripe dimension (k); (3) must have its expected failure reconstruction IO (AFR $\times$ k $\times$ disk-capacity) be no higher than was assumed possible for Rgroup0 (since disks in Rgroup0 are expected to have the highest AFR); (4) must have a recovery time in case of failure (MTTR) that does not exceed the maximum MTTR (set by the administrator when selecting the default redundancy scheme for Rgroup0).

Determining if a scheme is worth transitioning to. Whether the IO cost of transitioning to a scheme is worth it or not and what space-savings can be achieved by that transition is a function of the number of days disks will remain in that scheme (also known as *disk-days*). This, in turn, depends on (1) when the disks enter the new scheme, and (2) how soon disks will require another transition out of that scheme.

The time it takes for the disks to enter the new scheme is determined by the transition IO and the rate-limit. When the disks will transition out of the target Rgroup is dependent on the future and can only be estimated. For this estimation, the Rgroup-planner needs to estimate the number of days the AFR curve will remain below the threshold that forces a transition out. This needs different strategies for the two deployment patterns (trickle and step).

Recall that PACEMAKER knows the AFR curve for trickle-deployed disks (from the canaries) in advance. Recall that step-deployed disks have the property that the AFR curve learned from them is statistically robust and tends to exhibit gradual, as opposed to sudden AFR increases. The Rgroup-planner leverages these properties to estimate the future AFR behavior based on the recent past. Specifically, it takes the slope of the AFR curve in the recent past[4] and uses that to project the AFR curve rise in the future.

The number of disk-days in a scheme for it to be worth transitioning to is dictated by the IO constraints. For example, let us consider a disk running under PACEMAKER that requires a transition, and PACEMAKER is configured with an average-IO constraint of 1% and a peak-IO-cap of 5%. Suppose the disk requires 1 day to complete its transition at 100% IO bandwidth. With the current settings, PACEMAKER will only consider an Rgroup worthy of transitioning to (assuming it is allowed to use all 5% of its IO bandwidth) if at least 80 disk-days are spent after the disk entirely transitions to it (since transitioning to it would take up to 20 days at the allowed 5% IO bandwidth).

From among the viable schemes that are worth transitioning to based on the IO constraints, the Rgroup-planner chooses the one that provides the highest space-savings.

Decision on Rgroup creation. Rgroups cannot be created arbitrarily. This is because every Rgroup adds *placement restrictions*, since all chunks of a stripe have to be stored on disks belonging to the same Rgroup. Therefore, Rgroup-planner creates a new Rgroup only when (1) the resulting placement pool created by the new Rgroup is large enough to overcome traditional placement restrictions such as "no two chunks on the same rack[5]", and (2) the space-savings achievable by the chosen redundancy scheme is sufficiently greater than using an existing (less-space-efficient) Rgroup.

The disk deployment pattern also affects Rgroup formation. While the rules for whether to form an Rgroup remain the same for trickle and step-deployed disks, mixing disks deployed differently impacts the transitioning techniques that can be used for eventually transitioning disks out of that Rgroup. This in turn affects how the IO constraints are enforced. Specifically, for trickle deployments, creating an Rgroup for each set of transitioning disks would lead to too many small-sized Rgroups. So, for trickle-deployments, the Rgroup-planner creates a new Rgroup for a redundancy scheme if and only if one does not exist already. Creating Rgroups this way will also ensure that enough disks (thousands) will go into it to satisfy placement restrictions. Mixing disks from different trickle-deployments in the same Rgroup does not impact the IO constraints, because PACEMAKER optimizes the transition mechanism for few disks transitioning at a time, as is explained in §5.3. For step-deployments, due to the large fraction of disks that undergo transition together, having disks from multiple steps, or mixing trickle-deployed disks within the same Rgroup, creates adverse interactions (discussed in §5.3). Hence, the Rgroup-planner creates a new Rgroup for each step-deployment, even if there already exists one or more Rgroups that employ the chosen scheme. Each such Rgroup will contain many thousands of disks to overcome traditional placement restrictions. Per-step Rgroups also extend to the Rgroup with default redundancy schemes, implying a per-step Rgroup0. Despite having clusters with disk populations as high as 450K disks, PACEMAKER's restrained Rgroup creation led to no cluster ever having more than 10 Rgroups.

Rules for purging an Rgroup. An Rgroup may be purged for having too few disks. This can happen when too many of its constituent disks transition to other Rgroups, or they fail, or they are decommissioned leading to difficulty in fulfilling placement restrictions. If the Rgroup to be purged is

[4] PACEMAKER uses a 60 day (configurable) sliding window with an Epanechnikov kernel, which gives more weight to AFR changes in the recent past [21].

[5] Inter-cluster fault tolerance remains orthogonal to and unaffected by PACEMAKER.

made up of trickle-deployed disks, the Rgroup-planner will choose to RUp transition disks to an existing Rgroup with higher redundancy while meeting the IO constraints. For step-deployments, purging implies RUp transitioning disks into the more-failure-tolerant RGroup (RGroup0) that may include trickle-deployed disks.

5.3 Transition-executor

The transition-executor's role is to determine *how to transition the disks*. This involves choosing (1) the most IO-efficient technique to execute that transition, and (2) how to rate-limit the transition at hand. Once the transition technique is chosen, the transition-executor executes the transition via the rate-limiter as shown in Fig. 3.

Selecting the transition technique. Suppose the data needs to be conventionally re-encoded from a k_{cur}-of-n_{cur} scheme to a k_{new}-of-n_{new} scheme. The IO cost of conventional re-encoding involves reading–re-encoding–writing all the stripes whose chunks reside on each transitioning disk. This amounts to a read IO of $k_{cur}\times$disk-capacity (assuming almost-full disks), and a write IO of $k_{cur}\times$disk-capacity$\times\frac{n_{new}}{k_{new}}$ for a total IO $> 2\times k_{cur}\times$disk-capacity for each disk.

In addition to conventional re-encoding, PACEMAKER supports two new approaches to changing the redundancy scheme for disks and selects the most efficient option for any given transition. The best option depends on the fraction of the Rgroup being transitioned at once.

Type 1 (Transition by emptying disks). If a small percentage of an Rgroup's disks are being transitioned, it is more efficient to retain the contents of the transitioning disks in that Rgroup rather than re-encoding. Under this technique, the data stored on transitioning disks are simply moved (copied) to other disks within the current Rgroup. This involves reading and writing (elsewhere) the contents of the transitioning disks. Thus, the IO of transitioning via Type 1 is at most $2\times$disk-capacity, independent of scheme parameters, and therefore at least $k_{cur}\times$ cheaper than conventional re-encoding.

Type 1 can be employed whenever there is sufficient free space available to move the contents of the transitioning disks into other disks in the current Rgroup. Once the transitioning disks are empty, they can be removed from the current Rgroup and added to the new Rgroup as "new" (empty) disks.

Type 2 (Bulk transition by recalculating parities). If a large fraction of disks in an Rgroup need to transition together, it is more efficient to transition the entire Rgroup rather than only the disks that need a transition at that time. Most cluster storage systems use systematic codes[6] [8, 13, 14, 36], wherein transitioning an entire Rgroup involves only calculating and storing new parities and deleting the old parities. Specifically, the data chunks have to be only read for computing the new parities, but they do not have to be re-written. In contrast, if only a part of the disks are transitioned, some fraction of the data chunks also need to be re-written. Thus, the IO cost for transitioning via Type 2 involves a read IO of $\frac{k_{cur}}{n_{cur}}\times$disk-capacity, and a write IO of only the new parities, which amounts to a total IO of $\frac{n_{new}-k_{new}}{k_{new}}\times\frac{k_{cur}}{n_{cur}}\times$disk-capacity for each disk in the Rgroup. This is at most $2\times\frac{k_{cur}}{n_{cur}}\times$disk-capacity, which makes it at least $n_{cur}\times$ cheaper than conventional re-encoding.

Selecting the most efficient approach for a transition. For any given transition, the transition-executor selects the most IO-efficient of all the viable approaches. Almost always, trickle-deployed disks use Type 1 because they transition a-few-at-a-time, and step-deployed disks use Type 2 because Rgroup-planner maintains each step in a separate Rgroup.

Choosing how to rate limit a transition. Irrespective of the transitioning techniques, the transition-executor has to resolve the competing concerns of maximizing space-savings and minimizing risk of data loss via fast transitions, and minimizing foreground work interference by slowing down transitions so as to not overwhelm the foreground IO. Arbitrarily slowing down a transition to minimize interference is only possible when the transition is not in response to a rise in AFR. This is because a rising AFR hints at the data being under-protected if not transitioned to a higher redundancy soon. In PACEMAKER, a transition without an AFR rise occurs either when disks are being RDn transitioned at the end of infancy, or when they are being RUp transitioned because the Rgroup they belong to is being purged. For all the other RUp transitions, PACEMAKER carefully chooses how to rate limit the transition.

Determining how much bandwidth to allow for a given transition could be difficult, given that other transitions may be in-progress already or may be initiated at any time (we do observe concurrent transitions in our evaluations). So, to ensure that the aggregate IO of all ongoing transitions conforms to the peak-IO-cap cluster-wide, PACEMAKER limits each transition to the peak-IO-cap within its Rgroup. For trickle-deployed disks, which share Rgroups, the rate of transition initiations is consistently a small percentage of the shared Rgroup, allowing disk emptying to proceed at well below the peak-IO-cap. For step-deployed disks, this is easy for PACEMAKER, since a step only makes one transition at a time and its IO is fully contained in its separate Rgroup. The transition-executor's approach to managing peak-IO on a per-Rgroup basis is also why the proactive-transition-initiator can safely assume a rate-limit of the peak-IO-cap without consulting the transition-executor. If there is a sudden AFR increase that puts data at risk, PACEMAKER is designed to ignore its IO constraints to continue meeting the reliability constraint—this safety valve was never needed for any cluster evaluated.

After finalizing the transitioning technique, the transition-executor performs the necessary IO for transitioning disks (read, writes, parity recalculation, etc.). We find that the components required for the transition-executor are already

[6] In systematic codes, the data chunks are stored in unencoded form. This helps to avoid having to decode for normal (i.e., non-degraded-mode) reads.

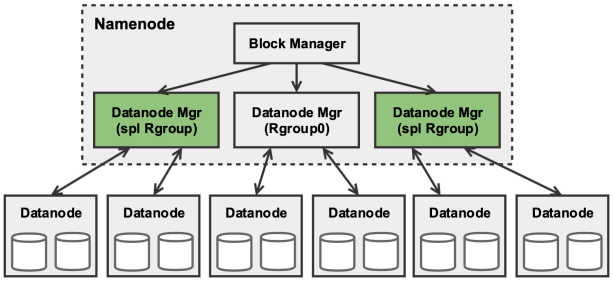


Figure 4: PACEMAKER-enhanced HDFS architecture.

present and adequately modular in existing distributed storage systems. In §6, we show how we implement PACEMAKER in HDFS with minimal effort.

Note that this design is for the common case where storage clusters are designed for a single dedicated storage service. Multiple distinct distributed storage services independently using the same underlying devices would need to coordinate their use of bandwidth (for their non-transition related load as well) in some way, which is outside the scope of this paper.

6 Implementation of PACEMAKER in HDFS

We have implemented a prototype of PACEMAKER for the Hadoop distributed file system (HDFS) [56]. HDFS is a popular open source distributed file system, widely employed in the industry for storing large volumes of data. We use HDFS v3.2.0, which natively supports erasure coding. Prototype of HDFS with Pacemaker is open-sourced and is available at https://github.com/thesys-lab/pacemaker-hdfs.git.

Background on HDFS architecture. HDFS has a central metadata server called Namenode (NN, akin to the master node) and a collection of servers containing the data stored in the file system, called Datanodes (DN, akin to worker nodes). Clients interact with the NN only to perform operations on file metadata (containing a collection of the DNs that store the file data). Clients directly request the data from the DNs. Each DN stores data on its local drives using a local file system.

Realizing Rgroups in HDFS. This design makes a simplifying assumption that all disks belonging to a DN are of the same Dgroup and are deployed together (this could be relaxed easily). Under this simplifying assumption, conceptually, an Rgroup would consist of a set of DNs that need to be managed independent of other such sets of DNs as shown in Fig 4.

The NN maintains a DatanodeManager (DNMgr), which is a gateway for the NN to interact with the DNs. The DNMgr maintains a list of the DNs, along with their usage statistics. The DNMgr also contains a HeartBeatManager (HrtBtMgr) which handles the periodic keepalive heartbeats from DNs. A natural mechanism to realize Rgroups in HDFS is to have one DNMgr per Rgroup. Note that the sets of DNs belonging to the different DNMgrs are mutually exclusive. Implementing Rgroups with multiple DNMgrs has several advantages.

Right level of control and view of the system. Since the DNMgr resides below the block layer, when the data needs to be moved for redundancy adaptations, the logical view of the file remains unaffected. Only the mapping from HDFS blocks to DNs gets updated in the inode. The statistics maintained by the DNMgr can be used to balance load across Rgroups.

Minimizing changes to the HDFS architecture and maximizing re-purposing of existing HDFS mechanisms. This design obviates the need to change HDFS's block placement policy, since it is implemented at the DNMgr level. Block placement policies are notoriously hard to get right. Moreover, block placement decisions are affected by fault domains and network topologies, both of which are orthogonal to PACEMAKER's goals, and thus best left untouched. Likewise, the code for reconstruction of data from a failed DN need not be touched, since all of the reads (to reconstruct each lost chunk) and writes (to store it somewhere else) will occur within the set of nodes managed by its DNMgr. Existing mechanisms for adding / decommissioning nodes managed by the DNMgr can be re-purposed to implement PACEMAKER's Type 1 transitions (described below).

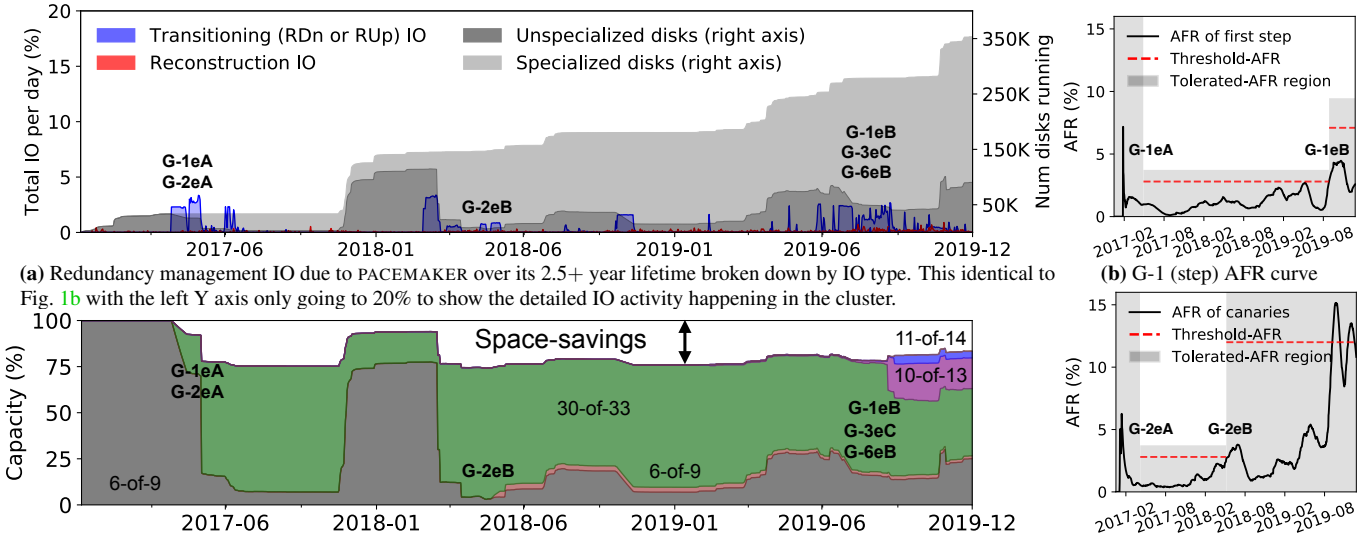
Cost of maintaining multiple DNMgrs is small. Each DNMgr maintains two threads: a HrtBtMgr and a DNAdminMgr. The former tracks and handles heartbeats from each DN, and the latter monitors the DNs for performing decommissioning and maintenance. The number of DNMgr threads in the NN will increase from two to 2× the number of Rgroups. Fortunately, even for large clusters, we observe that the number of Rgroups would not exceed the low tens (§7.4). The NN is usually a high-end server compared to the DNs, and an additional tens of threads shouldn't affect performance.

Rgroup transitions in HDFS. An important part of PACEMAKER functionality is transitioning DNs between Rgroups. Recall from §5.3 that one of PACEMAKER's preferred way of transitioning disks across Rgroups is by emptying the disks. In HDFS, the planned removal of a DN from a HDFS cluster is called decommissioning. PACEMAKER re-uses decommissioning to remove a DN from the set of DNs managed by one DNMgr and then adds it to the set managed by another, effectively transitioning a DN from one Rgroup to another.

PACEMAKER does not change the file manipulation API or client access paths. But, there is one corner-case related to transitions when file reads can be affected internally. To read a file, a client queries the NN for the inode and caches it. Subsequently, the reads are performed directly from the client to the DN. If the DN transitions to another Rgroup while the file is still being read, the HDFS client may find that that DN no longer has the requested data. But, because this design uses existing HDFS decommissioning for transitions, the client software knows to react by re-requesting the updated inode from the NN and resuming the read.

7 Evaluation

PACEMAKER-enabled disk-adaptive redundancy using is evaluated on production logs from four large-scale real-world storage clusters, each with hundreds of thousands of disks.



(a) Redundancy management IO due to PACEMAKER over its 2.5+ year lifetime broken down by IO type. This identical to Fig. 1b with the left Y axis only going to 20% to show the detailed IO activity happening in the cluster.

(b) G-1 (step) AFR curve

(c) Space-savings due to PACEMAKER. Each colored region represents the fraction of cluster capacity that is using a particular redundancy scheme. 6-of-9 is the default redundancy scheme (Rgroup0's).

(d) G-2 (trickle) AFR curve

Figure 5: Detailed IO analysis and space savings achieved by PACEMAKER-enabled adaptive redundancy on Google Cluster1.

We also experiment with a proof-of-concept HDFS implementation on a smaller sized cluster. This evaluation has four primary takeaways: (1) PACEMAKER eliminates transition overload, never using more than 5% of cluster IO bandwidth (0.2–0.4% on average) and always meets target MTTDL, in stark contrast to prior work approaches that do not account for transition IO load; (2) PACEMAKER provides more than 97% of idealized-potential space-savings, despite being proactive, reducing disk capacity needed by 14–20% compared to one-size-fits-all; (3) PACEMAKER's behavior is not overly sensitive across a range of values for its configurable parameters; (4) PACEMAKER copes well with the real-world AFR characteristics explained in §3.2. For example, it successfully combines the "multiple useful life phases" observation with efficient transitioning schemes. This evaluation also shows PACEMAKER in action by measuring disk-adaptive redundancy in PACEMAKER-enhanced HDFS.

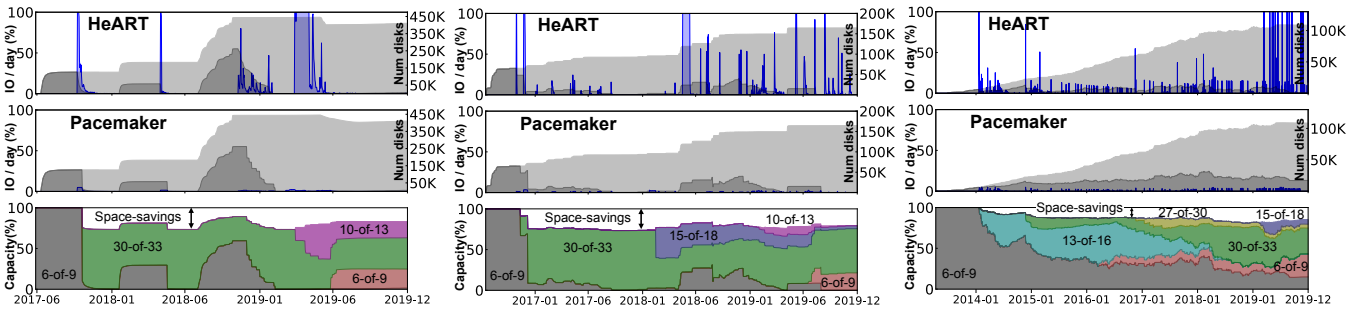
Evaluation methodology. PACEMAKER is simulated chronologically for each of the four cluster logs described in §3: three clusters from Google and one from Backblaze. For each simulated date, the simulator changes the cluster composition according to the disk additions, failures and decommissioning events in the log. PACEMAKER is provided the log information, as though it were being captured live in the cluster. IO bandwidth needed for each day's redundancy management is computed as the sum of IO for failure reconstruction and transition IO requested by PACEMAKER, and is reported as a fraction of the configured cluster IO bandwidth (100MB/sec per disk, by default).

PACEMAKER was configured to use a peak-IO-cap of 5%, an average-IO constraint of 1% and a threshold-AFR of 75% of the tolerated-AFR, except for the sensitivity studies in §7.3. For comparison, we also simulate (1) an idealized disk-adaptive redundancy system in which transitions are instantaneous (requiring no IO) and (2) the prior state-of-the-art approach (HeART) for disk-adaptive redundancy. For all cases, Rgroup0 uses 6-of-9, representing a one-size-fits-all scheme reported in prior literature [13]. The required target MTTDL is then back-calculated using the 6-of-9 default and an assumed tolerated-AFR of 16% for Rgroup0. These configuration defaults were set by consulting storage administrators of clusters we evaluated.

7.1 PACEMAKER on Google Cluster1 in-depth

Fig. 5a shows the IO generated by PACEMAKER (and disk count) over the ≈3-year lifetime of Google Cluster1. Over time, the cluster grew to over 350K disks comprising of disks from 7 makes/models (Dgroups) via a mix of trickle and step deployments. Fig. 5b and Fig. 5d show AFR curves of 2 of the 7 Dgroups (obfuscated as G-1 and G-2 for confidentiality) along with how PACEMAKER adapted to them at each age. G-1 disks are trickle-deployed whereas G-2 disks are step-deployed. The other 5 Dgroups are omitted due to lack of space. Fig. 5c shows the corresponding space-savings (the white space above the colors).

All disks enter the cluster as unspecialized disks, i.e. Rgroup0 (dark gray region in the Fig. 5a and left gray region of Figs. 5b and 5d). Once a Dgroup's AFR reduces sufficiently, PACEMAKER RDn transitions them to a specialized Rgroup (light gray area in Fig. 5a). Over their lifetime, disks may transition through multiple RUp transitions over the multiple useful life phases. Each transition requires IO, which is captured in blue in Fig. 5a. For example, the sudden drop in the unspecialized disks, and the blue area around 2018-04 captures the Type 2 transitions caused when over 100K disks RDn transition from Rgroup0 to a specialized Rgroup. The light gray region in Fig. 5a corresponds to the time over which space-savings are obtained, which can be seen in Fig. 5c.



(a) Google Cluster2 **(b)** Google Cluster3 **(c)** Backblaze

Figure 6: Top two rows show the IO overhead comparison between prior adaptive redundancy system (HeART) and PACEMAKER on two Google clusters and one Backblaze cluster. PACEMAKER successfully bounds all IO under 5% (visible as tiny blue regions in middle graphs, for e.g. around 2017 in (a)). The bottom row shows the 14–20% average space-savings achieved by PACEMAKER across the three clusters.

Many transitions with no transition overload. PACEMAKER successfully bounds all redundancy management IO comfortably under the configured peak-IO-cap throughout the cluster's lifetime. This can be seen via an imaginary horizontal line at 5% (the configured peak-IO-cap) that none of the blue regions goes above. Recall that PACEMAKER rate-limits the IO within each Rgroup to ensure simultaneous transitions do not violate the cluster's IO cap. Events ***G-1eA*** and ***G-2eA*** are examples of events where both G-1 and G-2 disks (making up almost 100% of the cluster at that time) request transitions at the same time. Despite that, the IO remains bounded below 5%. ***G-3eC*** and ***G-6eB*** also show huge disk populations of G-3 and G-6 Dgroups (AFRs not shown) requesting almost simultaneous RUp transitions, but PACEMAKER's design ensures that the peak-IO constraint is never violated. This is in sharp contrast with HeART's frequent transition overload, shown in Fig. 1a.

Disks experience multiple useful life phases. G-1, G-3, G-6 and G-7 disks experience two phases of useful life each. In Fig. 5a, events ***G-1eA*** and ***G-1eB*** mark the two transitions of G-1 disks through its multiple useful lives as shown in Fig. 5b. In the absence of multiple useful life phases, PACEMAKER would have RUp transitioned G-1 disks to Rgroup0 in 2019-05, eliminating space-savings for the remainder of their time in the cluster. §7.3 quantifies the benefit of multiple useful life phases for all four clusters.

MTTDL always at or above target. Along with the AFR curves, Figs. 5b and 5d also show the upper bound on the AFR for which the reliability constraint is met (top of the gray region). PACEMAKER sufficiently protects all disks throughout their life for all Dgroups across evaluated clusters.

Substantial space-savings. PACEMAKER provides 14% average space-savings (Fig. 5c) over the cluster lifetime to date. Except for 2017-01 to 2017-05 and 2017-11 to 2018-03, which correspond to infancy periods for large batches of new empty disks added to the cluster, the entire cluster achieves ≈20% space-savings. Note that the apparent reduction in space-savings from 2017-11 to 2018-03 isn't actually reduced space in absolute terms. Since Fig. 5c shows relative space-savings, the over 100K disks deployed around 2017-11, and their infancy period makes the space-savings appear reduced relative to the size of the cluster.

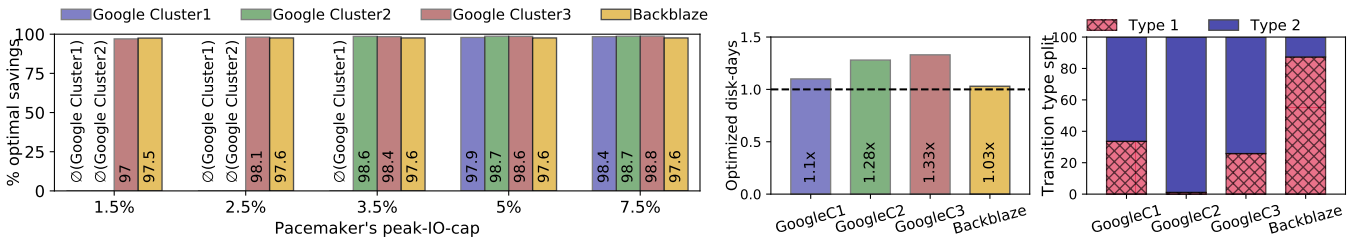
7.2 PACEMAKER on the other three clusters

Fig. 6 compares the transition IO incurred by PACEMAKER to that for HeART [27] for Google Cluster2, Google Cluster3 and Backblaze, along with the corresponding space-savings achieved by PACEMAKER. While clusters using HeART would suffer transition overload, the same clusters under PACEMAKER always had all their transition IO under the peak-IO-cap of 5%. In fact, on average, only 0.21–0.32% percent of the cluster IO bandwidth was used for transitions. The average space-savings for the three clusters are 14–20%.

Google Cluster2. Fig. 6a shows the transition overload and space-savings in Google Cluster2 and the corresponding space-savings. All Dgroups in Google Cluster2 are step-deployed. Thus, it is not surprising that Fig. 7c shows that over 98% of the transitions in Cluster2 were Type 2 transitions (bulk parity recalculation). Cluster2's disk population exceeds 450K disks. Even at such large scales, PACEMAKER obtains average space-savings of almost 17% and peak space-savings of over 25%. This translates to needing 100K fewer disks.

Google Cluster3. Google Cluster3 (Fig. 6b) is not as large as Cluster1 or Cluster2. At its peak, Cluster3 has a disk population of approximately 200K disks. But, it achieves the highest average space-savings (20%) among clusters evaluated. Like Cluster2, Cluster3 is also mostly step-deployed.

Backblaze Cluster. Backblaze (Fig. 6c) is a completely trickle-deployed cluster. The dark grey region across the bottom of Fig. 6c's PACEMAKER plot shows the persistent presence of canary disks throughout the cluster's lifetime. Unlike the Google clusters, the transition IO of Backblaze does not produce bursts of transition IO that lasts for weeks. Instead, since trickle-deployed disks transition a-few-at-a-time, we see transition work appearing continuously throughout the cluster lifetime of over 6 years. The rise in the transition IO spikes in 2019, for HeART, is because of large capacity 12TB disks replacing 4TB disks. Unsurprisingly, under PACEMAKER, most of the transitions are done using Type 1 (transitioning by emptying disks) as shown in Fig. 7c. The average space-savings obtained on Backblaze are 14%.



(a) PACEMAKER's sensitivity to the peak-IO constraint.

(b) Multiple useful life phases

(c) Transition type distribution

Figure 7: (a) shows PACEMAKER's sensitivity to the peak IO bandwidth constraint. (b) shows the advantage of multiple useful life phases and (c) shows the contribution of the two transitioning techniques when PACEMAKER was simulated on the four production clusters.

7.3 Sensitivity analyses and ablation studies

Sensitivity to IO constraints. The peak-IO constraint governs Fig. 7a, which shows the percentages of optimal space-savings achieved with PACEMAKER for peak-IO-cap settings between 1.5% and 7.5%. A peak-IO-cap of up to 7.5% is used in order to compare with the IO percentage spent for existing background IO activity, such as scrubbing. By scrubbing all data once every 15 days [5], the scrubber uses around 7% IO bandwidth, and is a background work IO level tolerated by today's clusters.

The Y-axis captures how close the space-savings are for the different peak-IO-caps compared to "Optimal savings", i.e. an idealized system with infinitely fast transitions. PACEMAKER's default peak-IO-cap (5%) achieves over 97% of the optimal space-savings for each of the four clusters. For peak-IO constraint set to <=2.5%, some RUp transitions in Google Cluster1 and Cluster2 become too aggressively rate-limited causing a subsequent AFR rise to violate the peak-IO constraints. We indicate this as a failure, and show it as "∅". The same situation happens for Google Cluster1 at 3.5%.

Sensitivity to threshold-AFR. The threshold-AFR determines when proactive RUp transitions of step-deployed disks are initiated. Conceptually, the threshold-AFR governs how risk-averse the admin wants to be. Lowering the threshold would trigger an RUp transition when disks are farther away from the tolerated-AFR (more risk-averse), and vice-versa. We evaluated PACEMAKER for threshold-AFRs of 60%, 75% and 90% of the respective Rgroups' tolerated-AFRs. We found that PACEMAKER's space-savings is not very sensitive to threshold-AFR, with space-savings only 2% lower at 60% than at 90%. Data remained safe at each of these settings, but would become unsafe with higher values.

Contribution of multiple useful life phases. Fig. 7b compares the increased number of disk-days spent in specialized Rgroups because of considering multiple useful life phases. In the best case, Google Cluster2 spent 33% more disk-days in specialized redundancy, increasing overall space-savings from 16% to 19%. Note that in large-scale storage clusters, even 1% space-savings are considered substantial as it represents thousands of disks.

Contribution of transition types. By proactively keeping step-deployed disks in distinct Rgroups and using specialized transitioning schemes whenever possible, instead of using simple re-encoding for all transitions, PACEMAKER reduces total transition IO by 92–96% for the four clusters. Fig. 7c shows what percentage of transitions were done via Type 1 (disk emptying) vs. Type 2 (bulk parity recalculation). As expected, Google clusters rely more on Type 2 transitions, because most disks are step-deployed. In contrast, the Backblaze cluster is entirely trickle-deployed and hence mostly uses Type 1 transitions. The small percentage of Type 2 transitions in Backblaze occur when Rgroups are purged.

7.4 Evaluating HDFS + PACEMAKER

This section describes basic experiments with the PACEMAKER-enabled HDFS, focusing on its functioning and operation. Note that PACEMAKER is designed for longitudinal disk deployments over several years, a scenario that cannot be reproduced identically in laboratory settings. Hence, these HDFS experiments are aimed to display that integrating PACEMAKER with an existing storage system is straightforward, rather than on the long-term aspects like overall space-savings or transition IO behavior over cluster lifetime as evaluated via simulation above.

The HDFS experiments run on a PRObE Emulab cluster [16]. Each machine has a Dual-Core AMD Opteron Processor, 16GB RAM, and Gigabit Ethernet. We use a 21-node cluster running HDFS 3.2.0 with one NN and 20 DNs. Each DN has a 10GB partition on a 10000 RPM HDD for a total cluster size of 200GB. We statically define the cluster to be made up of two Rgroups of ten DNs each, one using the 6-of-9 erasure coding scheme and the other using a 7-of-10 scheme. DFS-perf [19], a popular open-source HDFS benchmark is used, after populating the cluster to 60% full. Each DFS-perf client sequentially reads one file over and over again (size=768MB), for a total read size of about 1.75TB over 40 iterations. We use 60 DFS-perf clients, running on 20 nodes separate from the HDFS cluster.

We focus on the behavior of a DN as it transitions between Rgroups, compared with baseline HDFS performance (where all DNs are healthy) and its behavior while recovering from a failed DN. Fig. 8 shows the client throughput after the setup phase, followed by a noticeable drop in client throughput when a DN fails (emuated by stopping the DN). This is caused by the reconstruction IO that recreates the data from the failed node. Read latency exhibits similar behavior (not

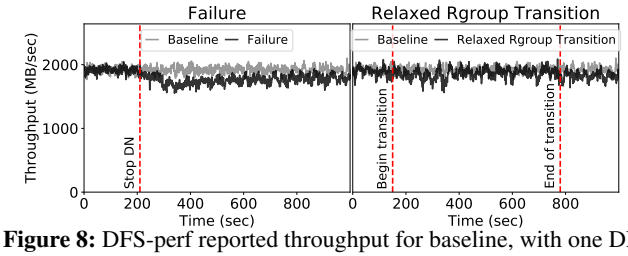


Figure 8: DFS-perf reported throughput for baseline, with one DN failure and one Rgroup transition.

shown due to space). Eventually, throughput settles at about 5% lower than prior to failure, since now there are 19 DNs.

Fig. 8 also shows client throughput when a node is RDn transitioned from 6-of-9 to 7-of-10. There is minor interference during the transition, which can be attributed to the data movement that HDFS performs as a part of decommissioning. The transition requires less work than failed node reconstruction, yet takes longer to complete because PACEMAKER limits the transition IO. Eventually, even though 20 DNs are running, the throughput is lower by ≈5% (one DN's throughput). This happens because PACEMAKER empties the DN before it moves into the new Rgroup, and load-balancing data to newly added DNs happens over a longer time-frame. Experiments with RUp transition showed similar results.

8 Related work

The closest related work [27] proposes a redundancy adaptation tool called HeART that categorizes disks into groups and suggests a tailored redundancy scheme for each during its useful life period. As discussed earlier, while [27] showcased potential space-savings, it ignored transition overload and hence is made impractical (Fig. 1a). PACEMAKER eliminates transition overload by employing IO constraints (specifically the peak-IO and average-IO constraints) that cap the transition IO to a tiny fraction cluster bandwidth. While HeART was evaluated only for the trickle-deployed Backblaze cluster, our evaluation of PACEMAKER for Google storage clusters exposes the unique challenges of step-deployed clusters. Several design elements were added to PACEMAKER to address the challenges posed by step-deployed disks.

Various systems include support for multiple redundancy schemes, allowing different schemes to be used for different data [12, 14]. Tools have been created for deciding, on a per-data basis, which scheme to use [59, 65]. Keeton et al. [28] describe a tool that automatically provides disaster-resistant solutions based on budget and failure models. PACEMAKER differs from such systems by focusing on efficiently adapting redundancy to different and time-varying AFRs of disks.

Reducing the impact of background IO, such as for data scrubbing, on foreground IO is a common research theme. [1, 3, 30, 31, 38, 53]. PACEMAKER converts otherwise-urgent bursts of transition IO into proactive background IO, which could then benefit from these works.

Disk reliability has been well studied, including evidence of failure rates being make/model dependent [5, 11, 22, 25, 32, 40, 41, 49–51, 55]. There are also studies that predict disk failures [2, 20, 33, 37, 58, 61, 68], which can enhance any storage fault-tolerance approach.

While several works have considered the problem of designing erasure codes that allow transitions using less resources, existing solutions are limited to specific kinds of transitions and hence are not applicable in general. The case of adding parity chunks while keeping the number of data chunks fixed can be viewed [35, 45, 47] as the well-studied reconstruction problem, and hence the codes designed for optimal reconstruction (e.g., [10, 18, 39, 46, 47, 60]) would lead to improved resource usage for this case. Several works have studied the case where the number of data nodes increases while the number of parity nodes remains fixed [23, 42, 64, 66, 69]. In [65], the authors propose two erasure codes designed to undergo a specific transition in parameters. In [34], the authors propose a general theoretical framework for studying codes that enable efficient transitions for general parameters, and derive lower bounds on the cost of transitions as well as describe optimal code constructions for certain specific parameters. However, none of the existing code constructions are applicable for the diverse set of transitions needed for disk-adaptive redundancy in real-world storage clusters.

9 Conclusion

PACEMAKER orchestrates disk-adaptive redundancy without transition overload, allowing use in real-world clusters. By proactively arranging data layouts and initiating transitions, PACEMAKER reduces total transition IO allowing it to be rate-limited. Its design integrates cleanly into existing scalable storage implementations, such as HDFS. Analysis for 4 large real-world storage clusters from Google and Backblaze show 14–20% average space-savings while transition IO is kept small (<0.4% on average) and bounded (e.g., <5%).

10 Acknowledgements

We thank our shepherd Wyatt Lloyd and the anonymous reviewers for their valuable feedback and suggestions. We extend special thanks to Larry Greenfield, Arif Merchant and numerous other researchers, engineers at Google; Keith Smith, Tim Emami, Jason Hennessey, Peter Macko and other researchers from NetApp's Advanced Technology Group (ATG) who have been instrumental in providing data, feedback and support. We also thank Jiaan Dai, Xuren Zhou, Jiaqi Zuo, Sai Kiriti Badam and Jiongtao Ye for their help in building the HDFS+PACEMAKER prototype. This research is generously supported in part by the NSF grants CNS 1956271 and CNS 1901410. We also thank the members and companies of the PDL Consortium (Alibaba, Amazon, Datrium, Facebook, Google, HPE, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, Two Sigma, Western Digital and VMware) for their interest, insights, feedback, and support.

References

[1] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[2] Preethi Anantharaman, Mu Qiao, and Divyesh Jadav. Large Scale Predictive Analytics for Hard Disk Remaining Useful Life Estimation. In *IEEE International Congress on Big Data (BigData Congress)*, 2018.

[3] Eitan Bachmat and Jiri Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *ACM SIGMETRICS Performance Evaluation Review*, 2002.

[4] Backblaze. Disk Reliability Dataset. https://www.backblaze.com/b2/hard-drive-test-data.html, 2013-2019.

[5] Lakshmi N Bairavasundaram, Garth R Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[6] Eric Brewer. Spinning Disks and Their Cloudy Future. https://www.usenix.org/node/194391, 2018.

[7] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for data centers. Technical report, Google, 2016.

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[9] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[10] Alexandros G. Dimakis, Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 2010.

[11] Jon Elerath. Hard-disk drives: The good, the bad, and the ugly. *Communication of ACM*, 2009.

[12] Erasure code Ceph Documentation. https://docs.ceph.com/docs/master/rados/operations/erasure-code/, (accessed Oct 15, 2020).

[13] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[14] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, (accessed Oct 15, 2020).

[15] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[16] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX; login*, 2013.

[17] Garth A Gibson. *Redundant disk arrays: Reliable, parallel secondary storage*. The MIT Press, 1992.

[18] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 2012.

[19] Rong Gu, Qianhao Dong, Haoyuan Li, Joseph Gonzalez, Zhao Zhang, Shuai Wang, Yihua Huang, Scott Shenker, Ion Stoica, and Patrick PC Lee. DFS-PERF: A scalable and unified benchmarking framework for distributed file systems. *UC Berkeley, Tech. Rep. UCB/EECS-2016-133*, 2016.

[20] Greg Hamerly and Charles Elkan. Bayesian approaches to failure prediction for disk drives. In *International Conference on Machine Learning (ICML)*, 2001.

[21] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Kernel smoothing methods. In *The elements of statistical learning*. Springer, 2009.

[22] Eric Heien, Derrick Kondo, Ana Gainaru, Dan LaPine, Bill Kramer, and Franck Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *ACM / IEEE High Performance Computing Networking, Storage and Analysis (SC)*, 2011.

[23] Yuchong Hu, Xiaoyang Zhang, Patrick P. C. Lee, and Pan Zhou. Generalized optimal storage scaling via network coding. In *IEEE International Symposium on Information Theory, ISIT*, 2018.

[24] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[25] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage (TOS)*, 2008.

[26] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Jungcheng Yang, K. V. Rashmi, and Gregory R. Ganger. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy (expanded). In *arXiv*, 2020.

[27] Saurabh Kadekodi, K V Rashmi, and Gregory R Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *USENIX File and Storage Technologies (FAST)*, 2019.

[28] Kimberly Keeton, Cipriano A Santos, Dirk Beyer, Jeffrey S Chase, John Wilkes, et al. Designing for disasters. In *USENIX File and Storage Technologies (FAST)*, 2004.

[29] Larry Lancaster and Alan Rowe. Measuring real-world data availability. In *USENIX LISA*, 2001.

[30] Christopher R Lumb, Jiri Schindler, Gregory R Ganger, et al. Freeblock scheduling outside of disk firmware. In *USENIX File and Storage Technologies (FAST)*, 2002.

[31] Christopher R Lumb, Jiri Schindler, Gregory R Ganger, David F Nagle, and Erik Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[32] Ao Ma, Rachel Traylor, Fred Douglis, Mark Chamness, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: characterizing, monitoring, and proactively protecting against disk failures. *ACM Transactions on Storage (TOS)*, 2015.

[33] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. Proactive error prediction to improve storage system reliability. In *USENIX Annual Technical Conference (ATC)*, 2017.

[34] Francisco Maturana and K. V. Rashmi. Convertible codes: new class of codes for efficient conversion of coded data in distributed storage. In *11th Innovations in Theoretical Computer Science Conference, ITCS*, 2020.

[35] Sara Mousavi, Tianli Zhou, and Chao Tian. Delayed parity generation in MDS storage codes. In *IEEE International Symposium on Info. Theory, ISIT*, 2018.

[36] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm BLOB storage system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[37] Joseph F Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado. Hard drive failure prediction using non-parametric statistical methods. In *Springer Artificial Neural Networks and Neural Information Processing (ICANN/CONIP*, 2003.

[38] Alina Oprea and Ari Juels. A Clean-Slate Look at Disk Scrubbing. In *USENIX File and Storage Technologies (FAST)*, 2010.

[39] Dimitris S. Papailiopoulos and Alexandros G. Dimakis. Locally repairable codes. *IEEE Transactions on Information Theory*, 2014.

[40] David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM International Conference on Management of Data (SIGMOD)*, 1988.

[41] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *USENIX File and Storage Technologies (FAST)*, 2007.

[42] Brijesh Kumar Rai, Vommi Dhoorjati, Lokesh Saini, and Amit K. Jha. On adaptive distributed storage systems. In *IEEE International Symposium on Information Theory, ISIT*, 2015.

[43] K V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[44] K V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[45] K. V. Rashmi, Nihar B. Shah, and P. Vijay Kumar. Enabling node repair in any erasure code for distributed storage. In *IEEE International Symposium on Information Theory Proceedings, ISIT*, 2011.

[46] K. V. Rashmi, Nihar B. Shah, and P. Vijay Kumar. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Trans. on Information Theory*, 2011.

[47] KV Rashmi, Nihar B Shah, and Kannan Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. *IEEE Transactions on Information Theory*, 2017.

[48] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. In *International Conference on Very Large Data Bases*, 2013.

[49] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding latent sector errors and how to protect against them. *ACM Trans. on Storage (TOS)*, 2010.

[50] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *USENIX File and Storage Technologies (FAST)*, 2007.

[51] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*. IOP Publishing, 2007.

[52] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *USENIX File and Storage Technologies (FAST)*, 2016.

[53] Thomas JE Schwarz, Qin Xin, Ethan L Miller, Darrell DE Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, 2004.

[54] Seagate. The Digitization of the World From Edge to Core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2018.

[55] Sandeep Shah and Jon G Elerath. Disk drive vintage and its effect on reliability. In *IEEE Reliability and Maintenance Symposium (RAMS)*, 2004.

[56] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[57] Emil Sit, Andreas Haeberlen, Frank Dabek, Byung-Gon Chun, Hakim Weatherspoon, Robert Tappan Morris, M Frans Kaashoek, and John Kubiatowicz. Proactive Replication for Data Durability. In *USENIX Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[58] Brian D Strom, SungChang Lee, George W Tyndall, and Andrei Khurshudov. Hard disk drive reliability modeling and failure prediction. *IEEE Transactions on Magnetics*, 2007.

[59] Eno Thereska, Michael Abd-El-Malek, Jay J Wylie, Dushyanth Narayanan, and Gregory R Ganger. Informed data distribution selection in a self-predicting storage system. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[60] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. Clay codes: Moulding {MDS} codes to yield an {MSR} code. In *USENIX File and Storage Technologies (FAST)*, 2018.

[61] Yu Wang, Eden WM Ma, Tommy WS Chow, and Kwok-Leung Tsui. A two-step parametric method for failure prediction in hard disk drives. *IEEE Trans. on industrial informatics*, 2014.

[62] Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*. Springer, 2002.

[63] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[64] Si Wu, Yinlong Xu, Yongkun Li, and Zhijia Yang. I/O-efficient scaling schemes for distributed storage systems with CRS codes. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[65] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A tale of two erasure codes in HDFS. In *USENIX File and Storage Technologies (FAST)*, 2015.

[66] Xiaoyang Zhang, Yuchong Hu, Patrick P. C. Lee, and Pan Zhou. Toward optimal storage scaling via network coding: from theory to practice. In *IEEE Conference on Computer Communications, INFOCOM*, 2018.

[67] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center. *Microsoft research MSR-TR-2010*, 52, 2010.

[68] Ying Zhao, Xiang Liu, Siqing Gan, and Weimin Zheng. Predicting disk failures with HMM-and HSMM-based approaches. In *Springer Industrial Conference on Data Mining (ICDM)*, 2010.

[69] Weimin Zheng and Guangyan Zhang. Fastscale: accelerate RAID scaling by minimizing data migration. In *USENIX File and Storage Technologies (FAST)*, 2011.



Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories

Jialin Li[1], Jacob Nelson[2], Ellis Michael[3], Xin Jin[4], and Dan R. K. Ports[2]

[1]National University of Singapore, [2]Microsoft Research, [3]University of Washington, [4]Johns Hopkins University

Abstract

High performance distributed storage systems face the challenge of load imbalance caused by skewed and dynamic workloads. This paper introduces Pegasus, a new storage system that leverages new-generation programmable switch ASICs to balance load across storage servers. Pegasus uses *selective replication* of the most popular objects in the data store to distribute load. Using a novel in-network coherence directory, the Pegasus switch tracks and manages the location of replicated objects. This allows it to achieve load-aware forwarding and dynamic rebalancing for replicated keys, while still guaranteeing data coherence and consistency. The Pegasus design is practical to implement as it stores only forwarding metadata in the switch data plane. The resulting system improves the throughput of a distributed in-memory key-value store by more than 10× under a latency SLO – results which hold across a large set of workloads with varying degrees of skew, read/write ratio, object sizes, and dynamism.

1 Introduction

Distributed storage systems are tasked with providing fast, predictable performance in spite of immense and unpredictable load. Systems like Facebook's memcached deployment [50] store trillions of objects and are accessed thousands of times on each user interaction. To achieve scale, these systems are distributed over many nodes; to achieve performance predictability, they store data primarily or entirely in memory.

A key challenge for these systems is balancing load in the presence of highly skewed workloads. Just as a celebrity may have many millions more followers than the average user, so too do some stored objects receive millions of requests per day while others see almost none [3, 67]. Moreover, the set of popular objects changes rapidly as new trends rise and fall [5]. While classic algorithms like consistent hashing [30] are effective at distributing load when all objects are of roughly equal popularity, here they fall short: requests for a single popular object commonly exceed the capacity of any individual server.

Replication makes it possible to handle objects whose request load exceeds one server's capacity. Replicating *every* object, while effective at load balancing [13, 49], introduces a high storage overhead. *Selective replication* of only a set of hot objects avoids this overhead. Leveraging prior analysis of caching [17], we show that surprisingly few objects need to be replicated in order to achieve strong load-balancing properties. However, keeping track of which objects are hot and where they are stored is not straightforward, especially when the storage system may have hundreds of thousands of clients, and keeping multiple copies consistent is even harder [50].

We address these challenges with Pegasus, a distributed storage system that uses a new architecture for selective replication and load balancing. Pegasus uses a programmable data-plane switch to route requests to servers. Drawing inspiration from CPU cache coherency protocols [4, 19, 22, 31, 34, 36, 37, 40], the Pegasus switch acts as an *in-network coherence directory* that tracks which objects are replicated and where. Leveraging the switch's central view of request traffic, it can forward requests to replicas in a load-aware manner. Unlike prior approaches, Pegasus's coherence directory also allows it to dynamically rebalance the replica set *on each write operation*, accelerating both read- and write-intensive workloads – while still maintaining strong consistency.

Pegasus introduces several new techniques, beyond the concept of the in-network coherence directory itself. It uses a lightweight version-based coherence protocol to ensure consistency. Load-aware scheduling is implemented using a combination of reverse in-network telemetry and in-switch weighted round-robin policy. Finally, to provide fault tolerance, Pegasus uses a simple chain replication [66] protocol to create multiple copies of data in different racks, each load-balanced with its own switch.

Pegasus is a practical approach. We show that it can be implemented using a Barefoot Tofino switch, and provides effective load balancing with minimal switch resource overhead. In particular, unlike prior systems [29, 45], Pegasus stores no application data in the switch, only metadata. This reduces switch memory usage to less than 3.5% of the total switch SRAM, permitting it to co-exist with existing switch functionality and thus reducing a major barrier to adoption [56].

Using 28 servers and a Pegasus switch, we show:

- Pegasus can increase the throughput by up to 10× – or reduce by 90% the number of servers required – of a system subject to a 99%-latency SLO.
- Pegasus can react quickly to dynamic workloads where the set of hot keys changes rapidly, and can recover quickly from server or rack failures.

- Pegasus can provide strong load balancing properties by only replicating a small number of objects.
- Pegasus is able to achieve these benefits for many classes of workloads, both read-heavy and write-heavy, with different object sizes and levels of skew.

2 Motivation

Real-world workloads for storage systems commonly exhibit highly skewed object access patterns [3, 6, 26, 50, 51]. Here, a small fraction of popular objects receive disproportionately more requests than the remaining objects. Many such workloads can be modeled using Zipfian access distributions [3, 5, 6, 67]; recent work has shown that some real workloads exhibit unprecedented skew levels (e.g., Zipf distributions with $\alpha > 1$) [10, 67]. Additionally, the set of popular objects changes dynamically: in some cases, the average hot object loses its popularity within 10 minutes [5].

Storage systems typically partition objects among multiple storage servers for scalability and load distribution. The implication of high skew in workloads is that load across storage servers is also uneven: the few servers that store the most popular objects will receive disproportionately more traffic than the others. The access skew is often high enough that the load for an object can exceed the processing capacity of a single server, leading to server overload. To reduce performance penalties, the system needs to be over-provisioned, which significantly increases overall cost.

Skewed workloads are diverse. Read-heavy workloads have been the focus of many recent studies, and many systems optimize heavily for them (e.g., assuming $> 95\%$ of requests are reads) [21, 29, 41, 45]. While many workloads do fall into this category, mixed or write-heavy workloads are also common [67]. Object sizes also vary widely, even within one provider. Systems may store small values (a few bytes), larger values (kilobytes to megabytes), or a combination of the two [1,3,5,67]. An ideal solution to workload skew should be able to handle all of these cases.

2.1 Existing Approaches

How should a storage system handle skewed workloads, where the request load for a particularly popular object might exceed the processing capability of an individual server? Two existing approaches have proven effective here: caching popular objects in a faster tier, and replicating objects to increase aggregate load capacity.

Caching Caching has long served as the standard approach for accelerating database-backed web applications. Recent work has demonstrated, both theoretically and practically, the effectiveness of a caching approach: only a small number of keys need to be cached in order to achieve provable load balancing guarantees [17, 29, 41].

There are, however, two limitations with the caching approach. First, the effectiveness of caching hinges on the ability to build a cache that can handle orders of magnitude more requests than the storage servers. Once an easily met goal, this has become a formidable challenge as storage systems themselves employ in-memory storage [50, 53, 58], clever data structures [42, 46], new NVM technologies [25, 68], and faster network stacks [38,42,48]. Recent efforts to build faster caches out of programmable switches [29, 45] address this, but hardware constraints impose significant limitations, e.g., an inability to support values greater than 128 bytes. Secondly, caching solutions only benefit read-heavy workloads, as cached copies must be invalidated until writes are processed by the storage servers.

Selective Replication Replication is another common solution to load imbalance caused by skewed workloads. By selectively replicating popular objects [2, 9, 13, 50], requests to these objects can be sent to any of the replicas, effectively distributing load across servers.

Existing selective replication approaches, however, face two challenges. First, clients must be able to identify the replicated objects and their locations – which may change as object popularity changes. This could be done using a centralized directory service, or by replicating the directory to the clients. Both pose scalability limitations: a centralized directory service can easily become a bottleneck, and keeping a directory synchronized among potentially hundreds of thousands of clients is not easy.

Providing consistency for replicated objects is the second major challenge – a sufficiently complex one that existing systems do not attempt to address it. They either replicate only read-only objects, or require users to explicitly manage inconsistencies resulting from replication [2, 9]. The solutions required to achieve strongly consistent replication (e.g., consensus protocols [35]) are notoriously complex, and incur significant coordination overhead [39], particularly when objects are modified frequently.

2.2 Pegasus Goals

The goal of our work is to provide an effective load balancing solution for the aforementioned classes of challenging workloads. Concretely, we require our system to 1) provide good load balancing for dynamic workloads with high skew, 2) work with fast in-memory storage systems, 3) handle arbitrary object sizes, 4) guarantee linearizability [24], and 5) be equally effective for read-heavy, write-heavy, and mixed read-/write workloads. As listed in Table 1, existing systems make explicit trade-offs and none of them simultaneously satisfy all five properties. In this paper, we will introduce a new distributed storage load balancing approach that makes no compromises, using an *in-network coherence directory*.

3 System Model

Pegasus is a design for rack-scale storage systems consisting of a number of storage servers connected via a single top-of-rack (ToR) switch, as shown in Figure 1. Pegasus combines in-switch load balancing logic with a new storage system. The

	Highly Skewed Workload	Fast In-Memory Store	All Object Sizes	Strong Consistency	Any Read-Write Ratio
Consistent Hashing [30]	✗	✓	✓	✗	–
Slicer [2]	✓	✗	✓	✗	–
Orleans [9]	✓	✗	✓	✗	–
EC-Cache [57]	✓	✗	✗	✓	✓
Scale-Out ccNUMA [21]	✓	✓	✓	✓	✗
SwitchKV [41]	✓	✗	✓	✓	✗
NetCache [29]	✓	✓	✗	✓	✗
Pegasus	✓	✓	✓	✓	✓

Table 1: A comparison of existing load balancing systems vs. Pegasus. In the "Any Read-Write Ratio" column, we only consider systems that provide strong consistency.

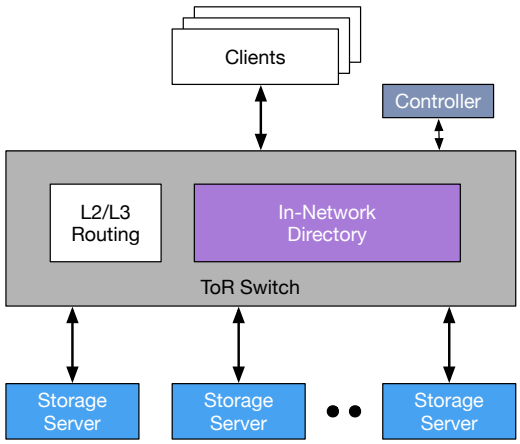


Figure 1: Pegasus system model. Pegasus is a rack-scale storage system. It augments the top-of-rack switch with an in-network coherence directory to balance load across storage servers in the rack. Servers store data in memory for fast and predictable performance.

Pegasus system provides a key-value store with a read/write interface. It does not support read-modify-write or atomic cross-key operations. Pegasus ensures strong data consistency (specifically, linearizability [24]). It uses in-memory storage to offer fast and predictable performance.

The Pegasus architecture is a co-design of in-switch processing and an application-level protocol. This is made possible by leveraging the capabilities of newly available switches with programmable dataplanes, such as the Barefoot Tofino, Cavium XPliant, or Broadcom Trident3 families. Broadly speaking, these chips offer reconfigurability in three relevant areas: (1) programmable parsing of application-specific headers; (2) flexible packet processing pipelines, usually consisting of 10–20 pipeline stages each capable of a match lookup and one or more ALU operations; and (3) general-purpose memory, on the order of 10 MB. Importantly, all of these features are on the switch dataplane, meaning that they can be used while processing packets at full line rate – a total capacity today measured in terabits per second.

Pegasus provides load balancing at the rack level, i.e., 32–256 servers connected by a single switch. It does not provide fault tolerance guarantees within the rack. Larger-scale, resilient systems can be built out of multiple Pegasus racks. For these systems, Pegasus ensures availability using a chain replication protocol to replicate objects across multiple racks for fault tolerance.

4 A Case for In-Network Directories

As we have discussed in §2, selectively replicating popular objects can offer good load balancing for highly skewed workloads, and it avoids certain drawbacks of a caching approach. Existing selective replication solutions, however, fall short in providing efficient directory services and strong consistency for the dynamic set of replicated objects. Our key observation is that in a rack-scale storage system (§3), the ToR switch serves as a *central point* of the system and is on the path of every client request and server reply. This enables us to implement a *coherence directory* abstraction in the ToR switch that addresses both challenges at the same time. It can track the location of every replicated object in the system and forward requests to servers with available capacity, and even change the number or location of replicas by determining where to send WRITE requests. Leveraging this in-network coherence directory, we co-design a *version-based coherence protocol* which guarantees linearizability and is highly efficient at processing object updates, enabling us to provide good load balancing even for write-intensive workloads.

4.1 Coherence Directory for Replicated Data

How do we design an efficient selective replication scheme that provides strong consistency? At a high level, the system needs to address the following challenges: first, it needs to track the replicated items and their locations with the latest value (i.e., the replica set). Second, read requests for a replicated object must be forwarded to a server in the current replica set. Third, after a write request is completed, all subsequent read requests must return the updated value.

The standard distributed systems approaches to this problem do not work well in this environment. One might try to have clients contact any server in the system, which then forwards the query to an appropriate replica for the data, as in distributed hash tables [14, 59, 60]. However, for in-memory storage systems, receiving and forwarding a request imposes

nearly as much load as executing it entirely. Nor is it feasible for clients to directly track the location of each object (e.g., using a configuration service [8, 27]), as there may be hundreds of thousands or millions of clients throughout the datacenter, and it is a costly proposition to update each of them as new objects become popular or an object's replica set is updated.

In Pegasus, we take a different approach. We note that these are the same set of challenges faced by CPU cache coherence and distributed shared memory systems. To address the above issues, these systems commonly run a cache coherence protocol using a coherence directory [4, 19, 22, 31, 34, 36, 37, 40]. For each data block, the coherence directory stores an entry that contains the set of processors that have a shared or exclusive copy. The directory is kept up to date as processors read and write blocks – invalidating old copies as necessary – and can always point a processor to the latest version.

A coherence directory can be applied to selective replication. It can track the set of replicated objects and forward read requests to the right servers, and it can ensure data consistency by removing stale replicas from the replica set. However, to use a coherence directory for a distributed storage system requires the directory to handle all client requests. Implemented on a conventional server, it will quickly become a source of latency and a throughput bottleneck.

4.2 Implementing Coherence Directory in the Network

Where should we implement a coherence directory that processes all client requests while not becoming a performance bottleneck? The ToR switch, as shown in Figure 1, provides a viable option for our targeted rack-scale storage systems. Switch ASICs are optimized for packet I/O: current generation switches can support packet processing at more than 10 Tb/s aggregate bandwidth and several billion packets per second [64, 65]. The programmable switches we target have a fixed-length reconfigurable pipeline, so any logic that fits within the pipeline can run at the switch's full line rate. Thus, implementing the coherence directory in the ToR switch for a rack-scale storage system will not become the bottleneck nor add significant latency, as it already processes all network traffic for the rack.

But can we implement a coherence directory efficiently in the ToR switch? To do so, two challenges have to be addressed. First, we need to implement all data structures and functional logic of a coherence directory in the switch *data plane*. We show that this is indeed possible with recent programmable switches: we store the replicated keys and their replica sets in the switch's memory, match and forward based on custom packet header fields (e.g. keys and operation types), and apply directory updating rules for the coherence protocol. We give a detailed description of our switch implementation in §8.

Second, the switch data plane has limited resources and many are already consumed by bread-and-butter switch functionality [56]. As the coherence directory tracks the replica set for each replicated object, the switch can only support a limited number of objects to be replicated. Our design meets this challenge. Interestingly, it is possible to achieve provable load balancing guarantees if we only replicate the most popular $O(n \log n)$ objects to all servers, where n is the *number of servers* (not keys) in the system (we give a more detailed analysis of this result in §4.5). Moreover, the coherence directory only stores small metadata such as key hashes and server IDs. For a rack-scale system with 32–256 servers, the size of the coherence directory is a small fraction of the available switch resources.

4.3 A Coherence Protocol for the Network

Designing a coherence protocol using an in-network coherence directory raises several new challenges. Traditional CPU cache coherence protocols can rely on an ordered and reliable interconnection network, and they commonly block processor requests during a coherence update. Switch ASICs have limited buffer space and therefore cannot hold packets indefinitely. Network links between ToR switches and servers are also unreliable: packets can be arbitrarily dropped, reordered, or duplicated. Many protocols for implementing ordered and reliable communication require complex logic and large buffering space that are unavailable on a switch.

We design a new *version-based*, *non-blocking* coherence protocol to address these challenges. The switch assigns a monotonically increasing version number to each write request and inserts it in the packet header. Servers store these version numbers alongside each object, and attach the version number in each read and write reply. The switch additionally stores a *completed* version number for each replicated object in the coherence directory. When receiving read or write replies (for replicated objects), the switch compares the version in the reply with the *completed* version in the directory. If the version number in the reply is higher, the switch updates the *completed* version number and resets the replica set to include only the source server. Subsequent read requests are then forwarded to the server with the new value. When more than one server has the latest value of the object, the version number in the reply can be equal to the *completed* version. In that case, we add the source server (if not already present) to the replica set so that subsequent read requests can be distributed among up-to-date replicas.

This protocol – which we detail in §6 – guarantees linearizability [24]. It leverages two key insights. First, all storage system requests and replies have to traverse the ToR switch. We therefore only need to update the in-network coherence directory to guarantee data consistency. This allows us to avoid expensive invalidation traffic or inter-server coordination overhead. Second, we use version numbers, applied by the switch to packet headers, to handle network asynchrony.

4.4 Load-Aware Scheduling

When forwarding read requests, the switch can pick any of the servers currently in the replica set. The simplest policy is to

select a random server from the set and rely on statistical load balancing among the servers. However, this approach falls short when the processing capacity is uneven on the storage servers (e.g. due to background tasks or different hardware specifications). To handle this issue, we also implement a weighted round-robin policy: storage servers periodically report their system load to the controller. The controller assigns weights for each server based on these load information and installs them on the switch. The switch then forwards requests to servers in the replica set proportional to their weights. Note that our in-network directory approach provides the mechanism for performing server selection. A full discussion of all scheduling policies is beyond the scope of this paper.

Surprisingly, these mechanisms can also be used for write requests. At first glance, it appears necessary to broadcast new writes to all servers in the replica set – potentially creating significant load and overloading some of the servers. However, the switch can choose a *new* replica set for the object on *each* write. It can forward write requests to one or more of the servers, and the coherence directory ensures data consistency, no matter which server the switch selects. The ability to move data frequently allows a switch to use load-aware scheduling *for both read and write requests*. This is key to Pegasus's ability to improve performance for both read- and write-intensive workloads.

4.5 Feasibility of An In-Network Coherence Directory

Pegasus makes efficient use of switch resources because it only tracks object metadata (vs. full object contents [29]), and only for a small number of objects. We claimed in §4.2 that Pegasus only needs to replicate the most popular $O(n \log n)$ objects (where n is the number of servers) to achieve strong load balancing guarantees. This result is an extension of previous work [17] which showed that *caching* the $O(n \log n)$ most frequently accessed objects is sufficient to achieve provable load balancing. That is, if we exclude these objects, the remaining load on each server exceeds the average load by at most a slack factor α, which depends on the constant factors but is generally quite small; see §9.5. Intuitively, most of the load in a highly-skewed workload is (by definition) concentrated in a few keys, so eliminating that load rebalances the system.

Our approach, rather than absorbing that load with a cache, is to redistribute it among the storage servers. A consequence of the previous result is that if the total request handling capacity of the system exceeds the request load by a factor of α, then there exists a way to redistribute the requests of the top $O(n \log n)$ keys such that no server exceeds its capacity. For read-only workloads, a simple way to achieve this is to replicate these keys to all servers, then route request to any server with excess capacity, e.g., by routing a request for a replicated key to the least-loaded server in the system.

Writes complicate the situation because they must be processed by all servers storing the object. As described in §4.4,



Figure 2: Pegasus packet format. The Pegasus application-layer header is embedded in the UDP payload. OP *is the request or reply type.* KEYHASH *contains the hash value of the key.* VER *is an object version number.* SERVERID *contains a unique server identifier.*

Pegasus can pick a new replica set, and a new replication factor, for an object on each write. Pegasus accomodates write-intensive workloads by tracking the write fraction for each object and setting the replication factor proportional to the expected number of reads per write, yielding constant overhead. Strictly speaking, our initial analysis (for read-only workloads) may not apply in this case, as it is no longer possible to send a read to *any* server. However, since Pegasus can rebalance the replica set on every write and dynamically adjusts the replication factor, it remains effective at load balancing for any read-write ratio. Intuitively, a read-mostly workload has many replicas, so Pegasus has a high degree of freedom for choosing a server for each read, whereas a write-mostly workload has fewer replicas but constantly rebalances them to be on the least-loaded servers.

5 Pegasus Overview

We implement an in-network coherence directory in a new rack-scale storage system, Pegasus. Figure 1 shows the high level architecture of a Pegasus deployment. All storage servers reside within a single rack. The top-of-rack (ToR) switch that connects all servers implements Pegasus's coherence directory for replicated objects.

Switch. The ToR switch maintains the coherence directory: it stores the set of replicated keys, and for each key, a list of servers with a valid copy of the data. To reduce switch resource overhead and to support arbitrary key sizes, the directory identifies keys by a small fixed-sized hash.

Pegasus defines an application-layer packet header embedded in the L4 payload, as shown in Figure 2. Pegasus reserves a special UDP port for the switch to match Pegasus packets. The application-layer header contains an OP field, either READ, WRITE, READ-REPLY, or WRITE-REPLY. KEYHASH is an application-generated, fixed-size hash value of the key. VER is an object version number assigned by the switch. SERVERID contains a unique identification of the server and is filled by servers on replies. If at-most-once semantics is required (§6.4), the header will additionally contain REQID, a globally unique ID for the request (assigned by the client). Non-Pegasus packets are forwarded using standard L2/L3 routing, keeping the switch fully compatible with existing network protocols.

To keep space usage low, the Pegasus switch keeps directory entries only for the small set of replicated objects. Read and write requests for replicated keys are forwarded according to the Pegasus load balancing and coherence protocol. The

Switch States:
- `ver_next`: next version number
- `rkeys`: set of replicated keys
- `rset`: map of replicated keys → set of servers with a valid copy
- `ver_completed`: map of replicated keys → version number of the latest completed WRITE

Figure 3: Switch states

other keys are mapped to a *home server* using a fixed algorithm, e.g., consistent hashing [30]. Although this algorithm could be implemented in the switch, we avoid the need to do so by having clients address their packets to the appropriate server; for non-replicated keys, the Pegasus switch simply forwards them according to standard L2/L3 forwarding policies.

Controller. The Pegasus control plane decides *which* keys should be replicated. It is responsible for updating the coherence directory with the most popular $O(n \log n)$ keys. To do so, the switch implements a request statistics engine that tracks the access rate of each key using both the data plane and the switch CPU. The controller – which can be run on the switch CPU, or a remote server – reads access statistics from the engine to find the most popular keys. The controller keeps only soft state, and can be immediately replaced if it fails.

6 Pegasus Protocol

To simplify exposition, we begin by describing the core Pegasus protocol (§6.2), under the assumption that the set of popular keys is fixed, and show that it provides linearizability. We then show how to handle changes in which keys are popular (§6.3), and how to provide exactly-once semantics (§6.4). Finally, we discuss server selection policies (§6.5) and other protocol details (§6.6).

Additionally, a TLA+ specification of the protocol which we have model checked for safety is available in our public repository [55].

6.1 Switch State

To implement an in-network coherence directory, Pegasus maintains a small amount of metadata in the switch dataplane, as listed in Figure 3. A counter `ver_next` keeps the next version number to be assigned. A lookup table `rkeys` stores the $O(n \log n)$ replicated hot keys, using KEYHASH in the packet header as the lookup key. For each replicated key, the switch maintains the set of servers with a valid copy in `rset`, and the version number of the latest completed WRITE in `ver_completed`. In §8, we elaborate how we store this state and implement this functionality in the switch dataplane.

6.2 Core Protocol: Request and Reply Processing

The core Pegasus protocol balances load by tracking the replica set of popular objects. It can load balance READ operations by choosing an existing replica to handle the request, and can change the replica set for an object by choosing which replicas process WRITE operations. Providing this load balancing while ensuring linearizability requires making sure that the in-network directory tracks the location of the *latest successfully written value* for each replicated key. Pegasus does this by assigning version numbers to incoming requests and monitoring outgoing replies to detect when a new version has been written.

Algorithm 1 HandleRequestPacket(pkt)

```
1:  if pkt.op = WRITE then
2:      pkt.ver ← ver_next++
3:  end if
4:  if rkeys.contains(pkt.keyhash) then
5:      if pkt.op = READ then
6:          pkt.dst ← select replica from rset[pkt.keyhash]
7:      else if pkt.op = WRITE then
8:          pkt.dst ← select from all servers
9:      end if
10: end if
11: Forward packet
```

Algorithm 2 HandleReplyPacket(pkt)

```
1:  if rkeys.contains(pkt.keyhash) then
2:      if pkt.ver > ver_completed[pkt.keyhash] then
3:          ver_completed[pkt.keyhash] ← pkt.ver
4:          rset[pkt.keyhash] ← set(pkt.serverid)
5:      else if pkt.ver = ver_completed[pkt.keyhash] then
6:          rset[pkt.keyhash].add(pkt.serverid)
7:      end if
8:  end if
9:  Forward packet
```

6.2.1 Handling Client Requests

The Pegasus switch assigns a version number to every WRITE request, by writing `ver_next` into its header and incrementing `ver_next` (Algorithm 1 line 1-3). It determines how to forward a request by matching the request's key hash with the `rkeys` table. If the key is not replicated, the switch simply forwards the request to the original destination – the home server of the key. For replicated keys, it forwards READ requests by choosing one server from the key's `rset`. For replicated WRITEs, it chooses one or more destinations from the set of all servers. In both cases, this choice is made according to the server selection policy (§6.5).

Storage servers maintain a version number for each key alongside its value. When processing a WRITE request, the server compares VER in the header with the version in the store, and updates both the value and the version number *only if* the packet has a *higher* VER. It also includes the version number read or written in the header of READ-REPLY and WRITE-REPLY messages.

6.2.2 Handling Server Replies

When the switch receives a READ-REPLY or a WRITE-REPLY, it looks up the reply's key hash in the switch `rkeys` table.

If the key is replicated, the switch compares VER in the packet header with the latest completed version of the key in `ver_completed`. If the reply has a higher version number, the switch updates `ver_completed` and resets the key's replica set to include only the source server (Algorithm 2 line 1-4). If the two version numbers are equal, the switch adds the source server to the key's replica set (Algorithm 2 line 5-7).

The effect of this algorithm is that write requests are sent to a new replica set which may or may not overlap with the previous one. As soon as one server completes and acknowledges the write, the switch directs all future read requests to it – which is sufficient to ensure linearizability. As other replicas also acknowledge the same version of the write, they begin to receive a share of the read request load.

6.2.3 Correctness

Pegasus provides linearizability [24]. The intuition behind this is that the Pegasus directory monitors all traffic, and tracks where the latest observed version of a key is located. As soon as any client sees a new version of the object – as indicated by a READ-REPLY or WRITE-REPLY containing a higher version number – the switch updates the directory to send future read requests to the server holding that version.

The critical invariant is that the Pegasus directory contains at least one address of a replica storing a copy of the latest write to be *externalized*, as well as a version number of that write. A write is externalized when its value can be observed outside the Pegasus system, which can happen in two ways. The way a write is usually externalized is when a WRITE-REPLY is sent, indicating that the write has been completed. It is also possible, if the previous and current replica set overlap, that a server will respond to a concurrent READ with the new version before the WRITE-REPLY is delivered. Pegasus detects both cases by monitoring both WRITE-REPLY and READ-REPLY messages, and updating the directory if VER exceeds the latest known compatible version number.

This invariant, combined with Pegasus's policy of forwarding reads to a server from the directory's replica set, is sufficient to ensure linearizability:

- WRITE operations can be ordered by their version numbers.
- If a READ operation r is submitted after a WRITE operation w completes, then r comes after w in the apparent order of operations because it is either forwarded to a replica with the version written by w or a replica with a higher version number.
- If a READ operation r_2 is submitted after another READ r_1 completes, then it comes after r_1 in the apparent order of operations, because it will either be forwarded to a replica with the version r_1 saw or a replica with a newer version.

6.3 Adding and Removing Replicated Keys

Key popularities change constantly. The Pegasus controller continually monitors access frequencies and updates the coherence directory with the most popular $O(n \log n)$ keys. We elaborate how access statistics are maintained in §8.

When a new key becomes popular, Pegasus must create a directory entry for it. The Pegasus controller does this by adding the key's home server to `rset`. It also adds a mapping for the key in `ver_completed`, associating it with `ver_next` -1, the largest version number that could have been assigned to a write to that key at the key's home server. Finally, the controller adds the key to `rkeys`. This process does not immediately move or replicate the object. However, later WRITE requests will be sent to a new (and potentially larger) replica set, with a version number necessarily larger than the one added to the directly. Once these newly written values are externalized, they will added to the directory as normal.

Transitioning a key from the replicated to unreplicated state is similarly straight-forward. The controller simply marks the switch's directory entry for transition. The next WRITE for that key is sent to its home server; once the matching WRITE-REPLY is received, the key is removed from the directory.

Read-only objects and virtual writes. The protocol above only moves an object to a new replica set (or back to its home node) on the next write. While this simplifies design, it poses a problem for objects that are read-only or modified infrequently. Conceptually, Pegasus addresses this by performing a write that does not change the object's value when an object needs to be moved. More precisely, the controller can force replication by issuing a *virtual write* to the key's home server, instructing it to increment its stored version number to the one in `ver_completed` and to forward that value to other replicas so that they can be added to `rset` and assist in serving reads.

6.4 Avoiding Duplicate Requests

At-most-once semantics, where duplicated or retried write requests are not reexecuted, are desirable. There is some debate about whether these semantics are required by linearizability or an orthogonal property [18, 28], and many key-value stores do not have this property. Pegasus accommodates both camps by optionally supporting at-most-once semantics.

Pegasus uses the classic mechanism of maintaining a table of the most recent request from each client [43] to detect duplicate requests. This requires that the same server process the original and the retried request, a requirement akin to "stickiness" in classic load balancers. A simple way to achieve this would be to send each write request initially to the object's home server. However, this sacrifices load balancing of writes.

We instead provide duplicate detection without sacrificing load balancing by noticing that it is not necessary for one server to see all requests for an object – only that a retried request goes to the same server that previously processed it. Thus, Pegasus forwards a request initially to a single *detector node* – a server deterministically chosen by the *request's* unique REQID, rather than the key's hash. It also writes into a packet header the other replicas, if any, that the request should be sent to. The detector node determines if the request is a duplicate; if not, it processes it and forwards the request to the

other selected servers.

Some additional care is required to migrate client table state when a key transitions from being unpopular to popular and vice versa. We can achieve this by pausing WRITEs to the key during transitions. When a new key becomes popular, the controller retrieves existing client table entries from the home server and propagates them to all servers. When a popular key becomes unpopular, it queries all servers to obtain their client tables, and sends their aggregation (taking the latest entry for each client) to the home server. Once this is complete, the system can resume processing WRITEs for that key.

6.5 Server Selection Policy

Which replica should be chosen for a request? This is a policy question whose answer does not impact correctness (i.e., linearizability) but determines how effective Pegasus is at load balancing. As described in §4.4, we currently implement two such policies. The first policy is to simply pick a random replica and rely on statistical load balancing. A more sophisticated policy is to use weighted round-robin: the controller assigns weights to each server based on load statistics it collects from the servers, and instructs the switch to select replicas with frequency proportional to the weights.

Write replication policy. Read operations are sent to exactly one replica. Write requests can be sent to one or more servers, whether they are in the current replica set or not. Larger replica set sizes improve load balancing by offering more options for future read requests, but increase the cost of write operations. For write-heavy workloads, increasing the write cost can easily negate any load balancing benefit.

As discussed in §4.5, the switch tracks the average READs per WRITE for each replicated object. By choosing a replication factor to be proportional to this ratio, Pegasus can bound the overhead regardless of the write fraction.

6.6 Additional Protocol Details

Hash collisions. The Pegasus coherence directory acts on small key hashes, rather than full keys. Should there be a hash collision involving a replicated key and a non-replicated key, requests for the non-replicated key may be incorrectly forwarded to a server that is not its home server. To deal with this issue, each server tracks the set of all currently replicated keys (kept up to date by the controller per §6.3). Armed with this information, a server can forward the improperly forwarded request to the correct home server. This request chaining approach has little performance impact: it only affects hash collisions *involving the small set of replicated keys*. Moreover, we only forward requests for the unreplicated keys which have low access rate. In the extremely rare case of a hash collision involving two of the $O(n \log n)$ most popular keys, Pegasus only replicates one of them to guarantee correctness.

Version number overflow. Version numbers must increase monotonically. Pegasus uses 64-bit version numbers, which makes overflow unlikely: it would require processing transactions at the full line rate of our switch for over 100 years. Extremely long-lived systems, or ones that prefer shorter version numbers, can use standard techniques for version number wraparound.

Garbage collection. When handling WRITEs for replicated keys, Pegasus does not explicitly invalidate or remove the old version. Although this does not impact correctness – the coherence directory forwards all requests to the latest version – retaining obsolete copies forever wastes storage space on servers. We handle this issue through garbage collection. The Pegasus controller already notifies servers about which keys are replicated, and periodically reports the last-completed version number. Each server, then, can detect and safely remove a key if it has an obsolete version, or if the key is no longer replicated (and the server is not the home node for that key).

7 Beyond a Single Rack

Thus far, we have discussed single-rack, single-switch Pegasus deployments. Of course, larger systems need to scale beyond a single rack. Moreover, the single-rack architecture provides no availability guarantees when servers or racks fail: while Pegasus replicates popular objects, the majority of objects still have just one copy. This choice is intentional, as entire-rack failures are common enough to make replicating objects within a rack insufficient for real fault tolerance.

We address both issues with a multi-rack deployment model where each rack of storage servers and its ToR switch runs a separate Pegasus instance. The workload is partitioned across different racks, and chain replication [66] is used to replicate objects to multiple racks. Object placement is done using two layers of consistent hashing. A global configuration service [8, 27] maps each range of the keyspace to a *chain* of Pegasus racks. Within each rack, these keys are mapped to servers as in §5. In effect, each key is mapped to a chain of servers, each server residing in a different rack.

We advocate this deployment model because it uses in-switch processing only in the ToR switches in each rack. The remainder of the datacenter network remains unmodified, and in particular it does not require any further changes to packet routing, which has been identified as a barrier to adoption for network operators [56]. A consequence is that it cannot load balance popular keys across different racks. Our simulations, however, indicate that this effect is negligible at all but the highest workload skew levels: individual servers are easily overloaded, but rack-level overload is less common.

Replication Protocol. As in the original chain replication, clients send WRITEs to the head server in the chain. Each server forwards the request to the next in the chain, until reaching the tail server, which then replies to the client. Clients send READs to the tail of the chain; that server responds directly to the client. In each case, if the object is a popular one in that rack, the Pegasus switch can redirect or replicate it.

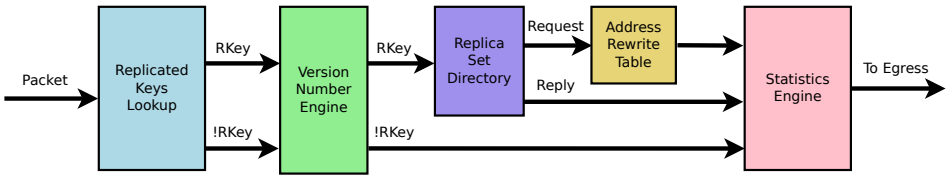


Figure 4: Switch data plane design for the Pegasus coherence directory

Pegasus differs from the original chain replication protocol in that it cannot assume reliable FIFO channels between servers. To deal with network asynchrony, it reuses the version numbers provided by the Pegasus switches to ensure consistency. Specifically, we augment Algorithm 1 in the following way: when a Pegasus switch receives a WRITE request, it only stamps `ver_next` into the request if VER in the packet header is not null; otherwise, it leaves the version number in the request unmodified and sets its `ver_next` to be one greater than that value (if it isn't already). The effect of this modification is that WRITE requests only carry version numbers from the head server's ToR switch; and the number does not change when propagating along the chain. This ensures that all replicas apply WRITEs in the same order.

Reconfiguring the Chains. If a Pegasus rack fails, it can be replaced using the standard chain replication protocol [66]. When the failure is noted, the configuration service is notified to remove the failed rack from all chains it participated in, and to add a replacement. This approach leverages the correctness of the underlying chain replication protocol, treating the Pegasus rack as functionally equivalent to a single replica.

If a system reconfiguration changes the identity of the head rack for a key range, subsequent WRITEs will get version numbers from a different head switch. If the new head rack was present in the old chain, these version numbers will necessarily be greater than any previously completed writes. If a rack is added to a chain as the head, the `ver_next` in the rack's switch must first be updated to be greater than or equal to the other switches in the chain.

If an individual server fails, a safe solution is to treat its entire rack as faulty and replace it accordingly. While correct, this approach is obviously inefficient. Pegasus has an optimized reconfiguration protocol (omitted due to space constraints) that only moves data that resided on the failed server.

8 Switch Implementation

The coherence directory (§6) plays a central role in Pegasus: it tracks popular objects and their replica sets; it distributes requests for load balancing; it implements the version-based coherence protocol; and it updates the set of replicated objects based on dynamic workload information. In this section, we detail the implementation of the Pegasus coherence directory in the data plane of a programmable switch.

8.1 Switch Dataplane Implementation

Figure 4 shows the schematic of the data plane design for the coherence directory. When a Pegasus packet enters the switch ingress pipeline, a lookup table checks if it refers to a replicated object. The packet then traverses a version number engine and a replica set directory, which implement the version-based coherence protocol (Algorithms 1 and 2). For request packets, one or more servers are selected from the replica set directory, and the packet's destination is updated by an address rewrite table. Finally, all Pegasus packets go through a statistics engine before being routed to the egress pipeline.

We leverage two types of stateful memory primitives available on programmable switching ASICs (such as Barefoot's Tofino [63]) to construct the directory: exact-match lookup tables and register arrays. A lookup table matches fields in the packet header and performs simple actions such as arithmetics, header rewrites, and metadata manipulations. Lookup tables, however, can only be updated from the control plane. Register arrays, on the other hand, are accessed by an *index* and can be read and updated at line rate in the data plane. The rest of the section details the design of each component.

Replicated Keys Lookup Table When adding replicated keys (§6.3), the controller installs its KEYHASH in an exact-match lookup table. The table only needs to maintain $O(n \log n)$ entries, where n is the number of servers in the rack. The switch matches each Pegasus header's KEYHASH with entries in the table. If there is a match, the index number associated with the entry is stored in the packet metadata to select the corresponding replicated key in later stages.

Version Number Engine We use two register arrays to build the version number engine, as shown in Figure 5. The first register array contains a single element – the next version number. If the packet is a WRITE request, the version number in the register is incremented and the switch writes the version into the packet header. The second register array stores the completed version number for each replicated key, and uses numeric ALUs to compare and update the version number (per Algorithm 2). The comparison result is passed to the next stage.

Replica Set Directory As shown in Figure 6, we build the replica set directory using three register arrays to store: (i) the size of each replica set, (ii) a bitmap indicating which servers are currently in each replica set, and (iii) a list of server IDs in each set. When choosing replicas for Pegasus READ requests,

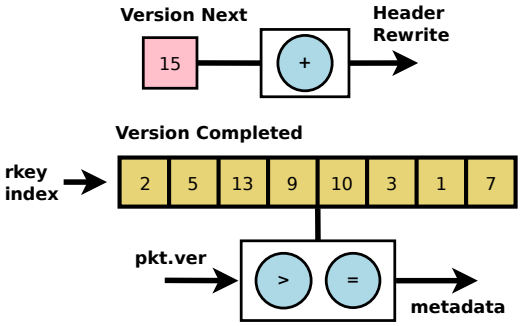


Figure 5: Design of the switch version number engine. One register array with a single element tracks the next version number. The register is incremented on each WRITE*. A second register array stores the latest completed version number for each replicated object. Numeric ALUs compare values in this array with version numbers in the reply headers.*

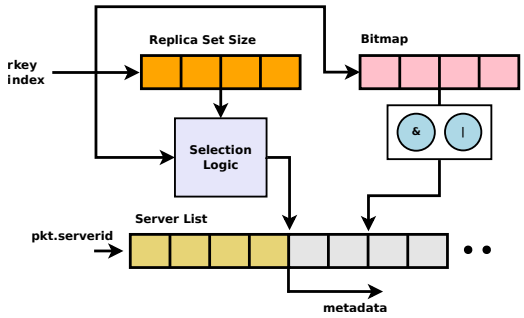


Figure 6: Design of the switch replica set directory. The directory uses three register arrays: one array stores the size of each replica set; another array maintains a bitmap for each set, tracking which servers are currently in the set; the last array stores a list of server IDs in each set.

the replica set size is read and fed into the selection logic unit to calculate an index for locating the server ID in the list (the selection logic can pick any servers for WRITE requests). Note that we collapse the server ID list of all replicated keys into a single register array, leveraging the fact that each key can be replicated on at most n servers. Therefore, to locate the ith replica for the kth key, the index is calculated as $k*n+i$ (for brevity, we will use relative indices, i.e. i in the formula, for the remaining discussion).

If the version number engine indicates that the Pegasus reply has a higher version number, the size of the replica set is reset to one, and the replica set bitmap and server ID list are reset to contain only the server ID in the reply packet. If the version number engine indicates that the version numbers are equal, the switch uses the bitmap to check if the server is already in the replica set. If not, it updates the bitmap, increments the replica set size, and appends the server ID to the end of the server list.

To add a new replicated key, the controller sets the replica set size to one, and the bitmap and server ID list to contain only the home server.

Address Rewrite Table The address rewrite table maps server IDs to the corresponding IP addresses and ports, and is kept up to date by the controller as servers are added. When the replica set directory chooses a single server as the destination, the address rewrite table updates the headers accordingly. If the replica set directory selects multiple servers (for a WRITE request), we use the packet replication engine on the switch to forward the packet to the corresponding multicast group.

Statistics Engine To detect the most popular $O(n\log n)$ keys in the workload, we construct a statistics engine to track the access rate of each key. For replicated keys, the switch maintains counters in a register array. This approach is obviously not feasible for the vast number of non-popular keys. The statistics engine instead samples requests for unreplicated keys, forwarding them to the switch CPU. A dedicated program on the switch CPU constructs an access frequency table from sampled packets. The sampling component serves two purposes: it reduces the traffic to the switch CPU and it functions as a high-pass filter to filter out keys with low access frequency. The controller scans both statistics tables to determine when newly popular keys need to be replicated or replication stopped for existing keys, and makes these changes following the protocol in §6.3.

Two separate register arrays track the READ and WRITE count for each replicated key. The controller uses these to compute the read/write ratio, which the selection logic in the replica set directory uses to decide how many replicas to use for each WRITE request.

9 Evaluation

Our Pegasus implementation includes switch data and control planes, a Pegasus controller, and an in-memory key-value store. The switch data plane is implemented in P4 [7] and runs on a Barefoot Tofino programmable switch ASIC [63]. The Pegasus controller is written in Python. It reads and updates the switch data plane through Thrift APIs [62] generated by the P4 SDE. The key-value store client and server are implemented in C++ with Intel DPDK [15] for optimized I/O performance.

Our testbed consists of 28 nodes with dual 2.2 GHz Intel Xeon Silver 4114 processors (20 total cores) and 48 GB RAM running Ubuntu Linux 18.04. These are connected to an Arista 7170-64S (Barefoot Tofino-based) programmable switch using Mellanox ConnectX-4 25 Gbit NICs. 16 nodes act as key-value servers and 12 generate client load.

To evaluate the effectiveness of Pegasus under realistic workloads, we generated load using concurrent open-loop clients, with inter-arrival time following a Poisson distribution. The total key space consists of one million randomly generated keys, and client requests chose keys following either a uniform distribution or a skewed (Zipf) distribution.

We compared Pegasus against two other load balancing solutions: a conventional static consistent hashing scheme for partitioning the key space, and NetCache [29]. The consis-

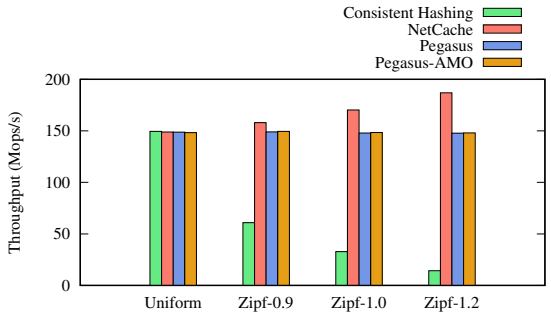


Figure 7: Maximum throughput achievable subject to a 99% latency SLO of 50 us. Pegasus successfully rebalances request load, maintaining similar performance levels for uniform and skewed workloads.

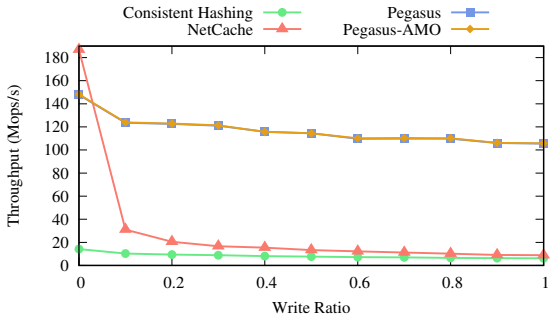


Figure 8: Throughput vs. write ratio. Pegasus maintains its load balancing advantage across the spectrum of write ratios, whereas NetCache suffers a significant penalty with even 10% writes.

tent hashing scheme assigns 16 virtual nodes to each storage server to improve load balancing. We additionally evaluated a version of Pegasus that supports at-most-once semantics (Pegasus-AMO, as described in §6.4). To allow a comparison with NetCache, we generally limit ourselves to 64-byte keys and 128-byte values, as this is the largest object value size it can support. NetCache reserves space for up to 10,000 128-byte values in the switch data plane, consuming a significant portion of the switch memory. In contrast, Pegasus consumes less than 3.5% of the total switch SRAM. At larger key and value sizes, Pegasus maintains similar performance and memory usage, whereas NetCache cannot run at all.

9.1 Impact of Skew

To test and compare the performance of Pegasus under a skewed workload, we measured the maximum throughput of all four systems subject to a 99%-latency SLO. We somewhat arbitrarily set the SLO to 5× of the median unloaded latency (we have seen similar results with different SLOs). Figure 7 shows system throughput under increasing workload skew with read-only requests. Pegasus maintains the same throughput level even as the workload varies from uniform to high to extreme skew (Zipf $\alpha = 0.9$–1.2),[1] demonstrating its effectiveness in balancing load under highly skewed access patterns. Since the workload is read-only, Pegasus with at-most-once support (Pegasus-AMO) has the exact same performance. In contrast, throughput of the consistent hashing system drops to as low as 10% under more skewed workloads. At $\alpha = 1.2$, Pegasus achieves a 10× throughput improvement over consistent hashing. NetCache provides similar load balancing benefits. In fact, its throughput *increases* with skew, outperforming Pegasus. This is because requests for the cached keys are processed directly by the switch, not the storage servers, albeit at the cost of significantly higher switch resource overhead.

[1] Although $\alpha = 1.2$ is a very high skew level, some major storage systems reach or exceed this level of skew. For example, more than half of Twitter's in-memory cache workloads can be modeled as Zipf distributions with $\alpha > 1.2$ [67], as can Alibaba's key-value store workload during peak usage periods [10].

9.2 Read/Write Ratio

Pegasus targets not only read-intensive workloads, but also write-intensive and read-write mixed workloads. Figure 8 shows the maximum throughput subject to a 99%-latency SLO of 50 μs when running a highly skewed workload (Zipf-1.2), with varying write ratio. The Pegasus coherence protocol allows write requests to be processed by any storage server while providing strong consistency, so Pegasus can load balance both read and write requests. As a result, Pegasus is able to maintain a high throughput level, regardless of the write ratio. Even with at-most-once semantics enforced, Pegasus-AMO performs equally well for all write ratios, by leveraging the randomness in requests' REQID (§6.4) to distribute write requests to all servers. This is in contrast to NetCache, which can only balance read-intensive workloads; it requires storage servers to handle writes. As a result, NetCache's throughput drops rapidly as the write ratio increases, approaching the same level as static consistent hashing. Even when only 10% of requests are writes, its throughput drops by more than 80%. Its ability to balance load is eliminated entirely for write-intensive workloads. In contrast, Pegasus maintains its high throughput even for write-intensive workloads, achieving as much as 11.8× the throughput as NetCache. Note that Pegasus's throughput does drop with higher write ratio. This is due to the increasing write contention and cache invalidation on the storage servers.

9.3 Scalability

To evaluate the scalability of Pegasus, we measured the maximum throughput subject to a 99%-latency SLO under a skewed workload (Zipf 1.2) with increasing number of storage servers, and compared it against the consistent hashing system. As shown in Figure 9, Pegasus scales nearly perfectly as the number of servers increases. On the other hand, throughput of consistent hashing stops scaling after two servers: due to severe load imbalance, the overloaded server quickly becomes the bottleneck of the entire system. Adding more servers thus does not further increase the overall throughput.

We also evaluate the performance of an end-host coherence directory implementation, using Pegasus's protocol with

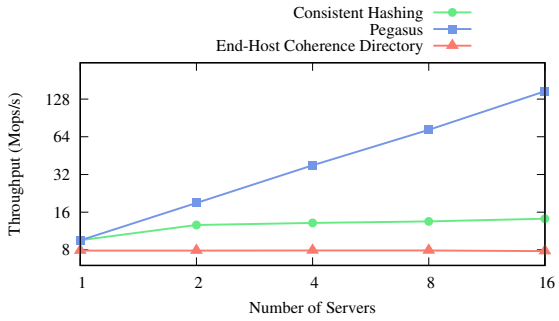


Figure 9: Scalability. Pegasus scales nearly linearly up to 16 servers, as no individual server becomes a bottleneck, even with a skewed workload.

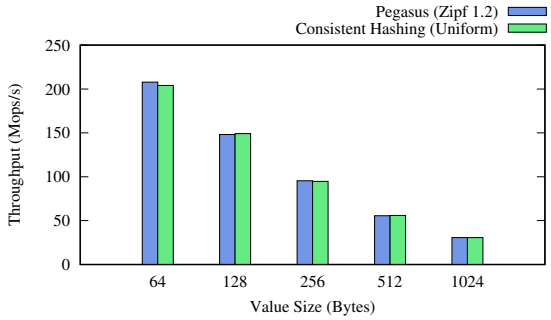


Figure 10: Throughput vs. object size. Pegasus provides effective load balancing across a wide range of object sizes. The performance of a traditional design under a uniform workload is shown as a baseline.

a server in place of the switch. Because the directory needs to process twice as many packets as the storage servers for each client request (both requests and replies), this implementation is unable to keep up with even a single server – highlighting the importance of using an accelerated platform like a switching ASIC as the coherence directory.

9.4 Object Sizes

To test if Pegasus can handle different object sizes, we varied the value size from 64 bytes to 1 KB and measured the maximum throughput of Pegasus subject to a 99%-latency SLO under the same skewed workload. We additionally plot the throughput of the consistent hashing system under a uniform workload. Figure 10 shows that Pegasus is equally efficient in load balancing for both small and large objects. Its throughput under a highly skewed workload is virtually equivalent to that of consistent hashing under a zero-skewed workload. Note that the throughput in the figure uses number of operations per second (which should naturally decrease with larger object size), not bits per second.

9.5 Impact of Number of Replicated Keys

Keeping the size of coherence directory small is crucial as switches are highly resource constrained. Our analysis (§4.5) shows that Pegasus only needs to replicate the $O(n \log n)$ most popular keys to balance load under arbitrary access patterns.

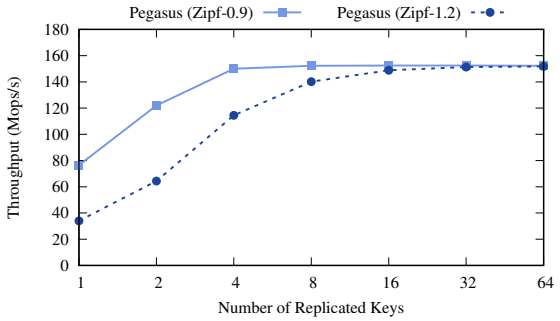


Figure 11: Throughput vs. number of replicated keys. For these workloads, only 8–16 replicated keys are needed to achieve most of Pegasus's load balancing benefit.

What constant factors are hidden here? For adversarial workloads, they are not high (e.g, $8n \log n$) [17]. We show in Figure 11 that they are even lower for our non-adversarial Zipf workload. Specifically, Pegasus only needs to replicate 8–16 keys to achieve its throughput benefit – significantly *less* than $n \log n$. While these numbers would be expected to increase with more servers, they easily remain within the capacity of the switch's register memory.

9.6 Server Selection Policies

We have implemented two policies for selecting servers for replicated objects: random and weighted round-robin. We evaluated both policies: Figure 12 shows their maximum throughput under different workloads.

Both policies are quite effective at distributing load for uniform and highly skewed workloads when we use a set of dedicated, homogeneous servers with the same load capacity. The random policy begins to fall short, however, when some servers are more capable than others, or background process sap their available capacity. We evaluated this by reducing the processing capacity of half of the servers by 50%. As shown in Figure 12, throughput with the random policy drops 50% as the slower servers become the performance bottleneck, even though the faster servers still have spare processing capacity. By collecting load information from the servers and setting the weights accordingly, the weighted round-robin policy allows both the slower and faster servers to fully utilize their processing capacity.

9.7 Handling Dynamic Workloads

Finally, we evaluated Pegasus under dynamic workloads with changing key popularity, similar to SwitchKV [41] and NetCache [29]. Specifically, we selected 100 keys every 10 seconds and changed their popularity rankings in the Zipf distribution. Here we consider two dynamic patterns:

- **Hot-in.** The 100 coldest keys in the popularity ranking are promoted to the top of the list, immediately turning them into the hottest objects. This workload represents extreme fluctuations in object popularities, which we hypothesize is rare in real world workloads.

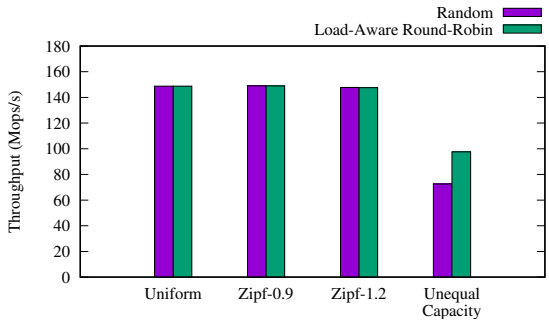
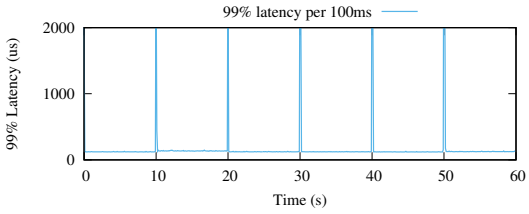
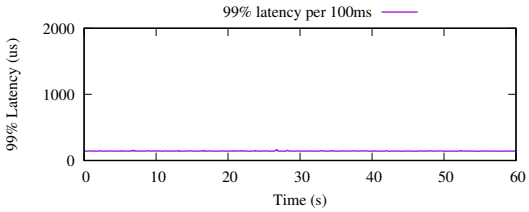


Figure 12: Comparing Pegasus server selection policies: throughput with a 99% latency SLO of 50 us. A random selection policy provides good statistical load balancing when server capacity is uniform; Pegasus's load-aware policy outperforms it otherwise.



(a) Hot-in: 100 unpopular keys become popular every 10 s



(b) Random: 100 random keys swap popularity every 10 s

Figure 13: Dynamic workloads. Pegasus reacts quickly to changes in object popularity.

- **Random.** We randomly select 100 keys from the 10,000 hottest keys, and swap their popularities with another set of randomly chosen keys. As the most popular keys are less likely to be changed, this dynamism represents a more moderate change to object popularity.

We evaluate Pegasus for these workloads with a Zipf-1.2 workload and 80% utilization.

Hot-in. Sudden changes to the popularity of *all* hottest keys cause the tail latency to increase. Pegasus, however, is able to immediately detect the popularity changes and updates the in-switch coherence directory. A workload change this drastic is unlikely, but Pegasus nevertheless reacts quickly. Within 100 ms, tail latency observed by clients returns to normal.

Random. Under a *random* dynamic pattern, only a moderate number of the most popular keys are changed. Pegasus thus can continue balancing load for the unaffected keys, and leveraging load-aware scheduling to avoid overloading the servers. No change in 99% end-to-end latency is observed.

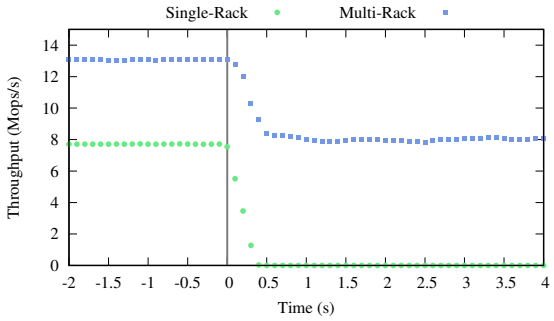


Figure 14: Throughput of single-rack vs. multi-rack configuration during a rack failure. After the failure ($t = 0$), the multi-rack configuration continues processing requests but loses some capacity.

9.8 Multi-Rack

To test a multi-rack configuration, we use a larger (but slower) cluster with 72 servers with dual 1.8 GHz Intel Xeon E5-2450 processors. These are organized into two racks, each with 24 storage servers and one Pegasus switch, plus a third rack of client machines. Per-node performance is significantly lower, largely because these servers use 10 Gbit NICs that do not support DPDK.

The two 24-server racks are configured into a 2-replica configuration: each rack acts as the head of the chain for half of the keys and the tail for the other half. Because both replicas need to handle WRITEs but only the tail processes READs, adding a second rack not only provides fault tolerance, it doubles read throughput; write throughput remains unchanged.

Figure 14 demonstrates this by comparing a single-rack and two-rack configurations, running a read-only workload with Zipf $\alpha = 1.2$; the two-rack configuration has 1.7× the throughput. At $t = 0$, one rack fails. The two-rack deployment is able to continue processing at half of its speed using the remaining rack. The single-rack deployment, of course, becomes entirely unavailable.

10 Related Work

Load Balancing. Load imbalance in large-scale key-value stores has been addressed by past systems in three ways. Consistent hashing [30] and virtual nodes [12] are widely used, but do not perform well with changing workloads. Solutions based on migration [11, 32, 61] and randomness [49] can be used to balance dynamic workloads, but these techniques introduce additional overheads and have limited ability to handle high skew. EC-Cache [57] balances load using erasure coding to split and replicate values, but works best for large keys in data-intensive clusters. SwitchKV [41] balances load across a flash-based storage layer using switches to route to an in-memory caching layer; it cannot react fast enough to changing load when the storage layer is in memory. NetCache [29] caches values directly in programmable dataplane switches; while this provides excellent throughput and latency, value sizes are limited by switch hardware constraints.

Another class of load balancers are designed to balance

layer 4 traffic, such as HTTP, across a dynamic set of backend servers. These systems may be implemented as clusters of servers, as in Ananta [54], Beamer [52], and Maglev [16]; or using switches, as in SilkRoad [47] or Duet [20]. These systems are designed to balance long-lived flows across servers, whereas Pegasus balances load of individual request packets. Prism [23] provides a way to perform request-level load balancing by migrating TCP and TLS connections, an approach that could be useful for Pegasus as an alternative to its UDP-based protocol.

Several new systems use programmable switches for application-specific load balancing protocols. R2P2 [33] load balances RPCs for stateless services where any request can be handled by any server. Harmonia [69] allows optimized forwarding for read requests in replicated systems by tracking when concurrent writes are in progress.

Directory-Based Coherence. Directory-based coherence protocols have been used in a variety of shared-memory multiprocessors and distributed shared memory systems [4, 19, 22, 31, 34, 36, 37, 40]. These systems can be thought of as key-value stores with fixed-size keys (addresses) and values (cache lines or pages). Directory protocols have been used in general key-value stores as well; IncBricks [44] implements an in-network key-value store using a distributed directory to cache values in network processors attached to datacenter switches. Keys have a designated home node that is involved in writes and coherence operations, limiting load-balancing opportunities for write-intensive workloads. Pegasus stores keys and values only in servers, and its coherence protocol allows any storage server to handle write requests, so Pegasus can load-balance both read- and write-intensive workloads. Both systems can scale beyond a rack and tolerate failures: IncBricks does so at the individual server level; Pegasus does so at the rack level.

11 Conclusion

With Pegasus, we have demonstrated that programmable switches can improve the load balancing of a storage application. Using our in-network coherence directory protocol, the switch takes over responsibility for placement of the most popular keys. This makes possible new data placement policies that cannot be achieved using traditional methods, such as reassigning the set of replicas on each write or selecting read replicas based on fine-grained load measurements. The end result is that Pegasus increases by $10\times$ the throughput level achievable subject to a latency SLO, compared to a consistent hashing workload. This permits a major reduction in the size of a cluster needed to support a particular workload.

More broadly, we believe that Pegasus provides an example of the class of applications that programmable dataplane switches are well suited for. It takes a classic use case for network devices – load balancing – and extends it to the next level by integrating it with an application-level protocol.

Acknowledgments

We thank our shepherd Simon Peter and the anonymous reviewers for their valuable feedback. This work was supported by NSF grant CNS-1615102 and gifts from Google and VMware. Jialin Li was supported by an MOE AcRF Tier 1 grant. Ellis Michael was supported by an IBM fellowship. Xin Jin was supported in part by NSF grants CNS-1813487, CCF-1918757 and CNS-1955487, a Facebook Communications & Networking Research Award, and a Google Faculty Research Award.

References

[1] A. Adya, R. Grandl, D. Myers, and H. Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019.

[2] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, Nov. 2016.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, England, UK, 2012.

[4] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*, 1990.

[5] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, Banff, AL, Canada, Nov. 2020.

[6] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, 2010.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, July 2014.

[8] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, Nov. 2006.

[9] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*, 2011.

[10] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li. HotRing: A hotspot-aware in-memory key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, USA, Feb. 2020.

[11] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*, Bordeaux, France, 2015.

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, AL, Canada, 2001.

[13] J. Dean and L. A. Barosso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, Oct. 2007.

[15] Intel Data Plane Development Kit. `https://www.dpdk.org/`.

[16] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI '16)*, Santa Clara, CA, 2016.

[17] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*, Cascais, Portugal, 2011.

[18] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the 12th ACM SIGOPS EuroSys (EuroSys '17)*, Belgrade, Serbia, Apr. 2017.

[19] S. Frank, H. Burkhardt, and J. Rothnie. The KSR 1: Bridging the gap between shared memory and MPPs. In *Digest of Papers. Compcon Spring*, pages 285–294, Feb 1993.

[20] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM SIGCOMM*, Chicago, IL, USA, 2014.

[21] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In *Proceedings of the 13th European Conference on Systems (Eurosys '18)*, 2018.

[22] D. B. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1):10–22, Jan. 1992.

[23] Y. Hayakawa, M. Honda, D. Santry, and L. Eggert. Prism: Proxies without the pain. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*, Boston, MA, USA, Feb. 2021.

[24] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, July 1990.

[25] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A network programming interface for non-volatile main memory. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, USA, Apr. 2018.

[26] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets '14)*, 2014.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, USA, June 2010.

[28] Incomplete high-level spec prevents verification of exactly-once semantic. `https://github.com/microsoft/Ironclad/issues/3`.

[29] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, 2017.

[30] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, El Paso, Texas, USA, 1997.

[31] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC '94, San Francisco, CA, USA, 1994.

[32] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper. The Yahoo!: Cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management (CloudDB '12)*, 2012.

[33] M. Koglas, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference*, Renton, WA, USA, July 2019.

[34] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA '94)*, 1994.

[35] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), Dec. 2001.

[36] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, 1997.

[37] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, Seattle, WA, USA, 1990.

[38] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, 2017.

[39] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, 2016.

[40] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[41] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI '16)*, Santa Clara, CA, USA, 2016.

[42] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, Apr. 2014.

[43] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[44] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, 2017.

[45] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. DistCache: Provable load balancing for large-scale storage systems with distributed caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, USA, Feb. 2019. USENIX.

[46] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*, Bern, Switzerland, Apr. 2012.

[47] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[48] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, USA, June 2013.

[49] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, Oct. 2001.

[50] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, 2013.

[51] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. An analysis of load imbalance in scale-out data serving. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS '16)*, 2016.

[52] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, 2018.

[53] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, Dec. 2009.

[54] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the 2013 ACM SIGCOMM*, SIGCOMM '13, Hong Kong, China, 2013.

[55] Pegasus public repository. `https://github.com/NUS-Systems-Lab/pegasus`.

[56] D. R. K. Ports and J. Nelson. When should the network be the computer? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019.

[57] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, Savannah, GA, USA, 2016.

[58] Redis in-memory data structure store. `https://redis.io/`.

[59] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, Nov. 2001.

[60] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):149–160, Feb. 2003.

[61] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, Nov. 2014.

[62] Apache Thrift software framework. `https://thrift.apache.org/`.

[63] Barefoot Tofino programmable switch ASICs. `https://www.barefootnetworks.com/products/brief-tofino/`.

[64] Barefoot Tofino 2: Second-generation of world's fastest P4-programmable Ethernet switch ASICs. `https://www.barefootnetworks.com/products/brief-tofino-2/`.

[65] Broadcom's Tomahawk 3 Ethernet switch chip.

[66] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI '04)*, San Francisco, CA, USA, 2004.

[67] J. Yang, Y. Yue, and R. Vinayak. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, Banff, AL, Canada, Nov. 2020.

[68] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, Mar. 2015.

[69] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment*, 13(3):376–389, Nov. 2019.

A Artifact Appendix

Abstract

Our artifact includes the following components: 1) P4 source code of the Pegasus switch data plane, 2) Python source code of the Pegasus switch controller, 3) C++ implementation of an in-memory key-value store with Intel DPDK, 4) configuration files and Python/shell scripts for running Pegasus experiments in a cluster, and 5) a TLA+ specification of the Pegasus protocol. The artifact is publicly available at: `https://github.com/NUS-Systems-Lab/pegasus`.

A.1 Artifact check-list

- **Algorithm:** Coherence protocol.
- **Program:** Key-value store, P4 packet processing program.
- **Compilation:** GCC 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04), Barefoot SDE 9.1.1
- **Binary:** Generated from GCC compiler and Barefoot SDE.
- **Run-time environment:** Ubuntu 18.04 LTS (Linux 4.15), Barefoot SDE 9.1.1
- **Hardware:** Dual socket 2.2 GHz Intel Xeon Silver 4114 processors with 20 cores and 48 GB RAM per socket. Mellanox ConnectX-4 25 Gbit NICs. Arista 7170-64S (barefoot Tofino-based) programmable switch.
- **Execution:** Bash and Python scripts.
- **Output:** Throughput. Average, median, 90%, 99% latencies.
- **Experiments:** Experiments as specified in the main paper (§9). Customizable experiment parameters: number of clients and servers, client request rate, read/write ratio, Zipfian coefficient, value size, number of keys, maximum number of replicated objects, and experiment duration.
- **Expected experiment run time:** 10-60 seconds per experiment.
- **Public link:** `https://github.com/NUS-Systems-Lab/pegasus`
- **Code licenses:** MIT license.

A.2 Description

A.2.1 How to access

All source code, configuration files, and scripts are publicly available at: `https://github.com/NUS-Systems-Lab/pegasus`.

A.2.2 Hardware dependencies

The artifact requires a P4 programmable switch (e.g., Barefoot Tofino programmable switch ASIC). The network interface cards on the client and server machines need to support Intel DPDK.

A.2.3 Software dependencies

The artifact has been tested on Ubuntu 18.04 LTS (Linux kernel 4.15). Compiling and running the Pegasus P4 data plane program require the Barefoot SDE (tested with version 9.1.1). Additional software package dependencies:

- libevent
- Intel TBB
- libnuma
- zlib
- DPDK (tested with version 19.11)
- Python Sorted Containers
- Python PyREM

A.2.4 Data sets

Experiments in this artifact expect a text file that contains ASCII keys (one key per line) for the key-value store. We provide a sample keys file, `artifact_eval/keys`, that has one million 64B-keys.

A.3 Installation

First, download or clone the repository. Throughout this document, we will use the following macros:

- `$REPO`: path to the root of the repository
- `$SDE`: path to Barefoot SDE
- `$SDE_INSTALL`: path to Barefoot SDE installation directory

A.3.1 Compiling Client and Server Code

Run `make` in `$REPO`.

A.3.2 Compiling P4 Code

On the target P4 switch:

```
cd $SDE/pkgsrc/p4-build
./configure P4_PATH=$REPO/p4/p4_tofino/pegasus.p4 \
            P4_NAME=pegasus P4_PREFIX=pegasus \
            P4_VERSION=p4-14 P4_FLAGS="--verbose 2" \
            --with-tofino --prefix=$SDE_INSTALL \
            --enable-thrift
make && make install
./configure P4_PATH=$REPO/p4/netcache/one.p4 \
            P4_NAME=netcache P4_PREFIX=netcache \
            P4_VERSION=p4-14 P4_FLAGS="--verbose 2" \
            --with-tofino --prefix=$SDE_INSTALL \
            --enable-thrift
make && make install
```

Note that the location of `p4-build` may depend on the Barefoot SDE version.

A.4 Experiment workflow

A.4.1 P4 Switch

First, start the Pegasus switch daemon on the P4 switch:

```
cd $SDE
./run_switchd.sh -p pegasus
```

Or if running NetCache, run the following:

```
cd $SDE
./run_switchd.sh -p netcache
```

In the switch shell, add and enable all switch ports used by the experiments.

Secondly, modify `$REPO/artifact_eval/pegasus.json` and `$REPO/artifact_eval/netcache.json` with the testbed cluster configuration (refer to `artifact_eval/README.md` for configuration file format).

Thirdly, start the Pegasus switch controller:

```
cd $REPO
./artifact_eval/run_pegasus_controller.sh
```

Or if running NetCache, run the following:

```
cd $REPO
./artifact_eval/run_netcache_controller.sh
```

A.4.2 End-Hosts

First, modify `$REPO/artifact_eval/testbed.config` with the cluster configuration. Refer to `artifact_eval/README.md` for the format of the file.

Secondly, modify the experiment python script `$REPO/artifact_eval/run_experiments.py`. Update `clients` and `servers` with actual host names of the client and server machines.

Lastly, on a machine that has ssh connectivity to all clients and servers, run the following:

```
python2 $REPO/artifact_eval/run_experiments.py
```

A.5 Evaluation and expected result

The experiment python script outputs the following statistics:

- Total throughput
- Average latency
- Median latency
- 90% latency
- 99% latency

Modify `n_client_threads` and `interval` in the experiment script to control the client load. Tune them until getting the maximum throughput with some 99% latency SLO, as reported in the paper.

To evaluate the different workloads and system configurations as specified in §9, vary the following parameters in the experiment script:

- `n_servers`: number of servers used in the experiment
- `node_config`: one of `pegasus`, `netcache`, or `static` (consistent hashing). Note that `pegasus` and `netcache` require running the corresponding P4 switch daemon and controller.
- `alpha`: Zipfian coefficient
- `get_ratio`: percentage of read requests in the workload (0.0 - 1.0)
- `key_type`: key access distribution. Either `unif` (uniform) or `zipf` (Zipfian)
- `value_len`: value size (in bytes)
- `n_keys`: total number of keys

A.6 AE Methodology

Submission, reviewing and badging methodology:

- `https://www.usenix.org/conference/osdi20/call-for-artifacts`

FlightTracker: Consistency across Read-Optimized Online Stores at Facebook

Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han,
Dmitri Petrov, Jim Carrig, John Hugg, Nathan Bronson
Facebook, Inc.

Abstract

Social media platforms deliver fresh personalized content by performing a large number of reads from an online data store. This store must be optimized for read efficiency, availability, and scalability. Multi-layer caches and asynchronous replication can satisfy these goals, such as in Facebook's graph store TAO, but it is challenging for the resulting system to provide a developer-friendly consistency model. TAO originally provided read-your-writes (RYW) consistency via write-through caching, but scaling challenges with this approach have led us to a new implementation.

This paper introduces FlightTracker, a family of APIs and systems which now manage consistency for online access to Facebook's graph. FlightTracker implicitly provides RYW and can be explicitly used to provide alternative consistency guarantees for special use cases; it enables flexible communication patterns between caches, which we have found important as the number of datacenters increases; it extends the same consistency guarantees to cross-shard indexes and materialized views, allowing us to transparently optimize queries; and it provides a uniform primitive for clients to obtain desired consistency guarantees across a variety of data stores. FlightTracker delivers these advantages while preserving the efficiency, latency, and availability benefits of asynchronous replication for the underlying systems, managing consistency for billions of users and more than 10^{15} queries per day.

1 Introduction

Social media platforms deliver fresh and customized aggregation of content. This feature combination makes it ineffective to aggregate ahead of time; instead each application-level web request at Facebook may issue hundreds or thousands of queries to our graph store TAO [20] to render a single response. This high query amplification means that data store reads must be efficient, low-latency, and highly available. At Facebook, we have addressed this challenge with an asynchronously coupled federation of caches, database replicas, and customized indexes that model social media data and metadata as a graph. While this read-optimized ecosystem achieves high performance, it is challenging to provide an intuitive and uniform consistency model to developers.

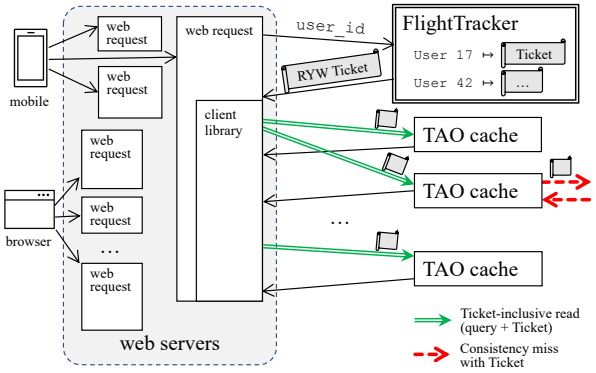


Figure 1: Web request flow with FlightTracker.

FlightTracker is our solution for managing RYW consistency for online access to the social graph at Facebook. It preserves the read efficiency, hot spot tolerance, and high availability of eventual consistency while providing RYW consistency. FlightTracker offers a uniform notion of an end user session that spans many stateful services and can be extended to new data stores without architecture changes.

FlightTracker consists of a family of APIs and a metadata service. Building on write-set tracking techniques [28, 40, 41, 51] and CRDTs [18, 50], the FlightTracker service accumulates the metadata of a user's recent writes and exposes the metadata as a data type we call a *Ticket*. Web requests fetch the user's Ticket once, as soon as the user is identified (see Figure 1). This Ticket is automatically attached on all subsequent queries to the social graph from the web request.

We use a variety of system-specific strategies to ensure that every write identified by the Ticket is reflected in query results. For example, our strategy for caches is to ignore cache entries that may be stale compared to writes in the Ticket; we refer to the resulting cache miss as a *consistency miss*. Systems can propagate Tickets recursively when they need to fetch data from another component while processing a query.

FlightTracker has been in production since 2016. It provides RYW consistency for billions of users and 10^{15} data store queries per day. For the majority of Facebook's internal applications and developers, FlightTracker is automatic and hidden. Some call sites and higher-layer infrastructure components explicitly manipulate Tickets to strengthen the consistency level. FlightTracker's loosely coupled design has allowed us to incrementally roll out support to two caching systems, three indexing systems, and two databases. It preserves the efficiency, latency, and availability that these data stores would offer under eventual consistency.

Overall, this paper makes five contributions:

- We summarize challenges Facebook encountered when relying on write-through caching for RYW in TAO, a read-optimized geo-replicated graph store (§ 2).
- We present the Ticket abstraction, which encapsulates the system-specific details of write sets in an extensible manner across service boundaries (§ 4).
- We show how we store and exchange Tickets in the FlightTracker service to provide RYW consistency (§ 3 and § 5) or explicitly satisfy alternative consistency requirements for select use cases (§ 7) while tolerating hot spots (§ 6.5).
- We explain a variety of strategies we used to implement Ticket-inclusive reads in query-serving systems (§ 6), including ones for simple caches and global indexes with complex update pipelines (§ 6.3).
- We evaluate FlightTracker in our production environment, demonstrating that it preserves the useful properties of the underlying read-optimized stores (§ 8), and share some lessons learned (§ 8.5).

2 Motivation

TAO is a read-optimized data store that provides access to the social graph at Facebook [20]. It is implemented using two layers of caches in front of a geo-replicated database. TAO originally relied on write-through caching for consistency. This technique provided RYW on top of eventual consistency, while preserving the read efficiency and hot spot tolerance of the system, since it allowed most queries to be served from a nearby L1 cache server.

As Facebook grew, we found that we needed a better approach to consistency. TAO's original write-through strategy relied on the use of a fixed communication pattern: users were made sticky to a single L1 cache cluster by the load balancers, inter-cluster communication was limited to traversing a fixed tree, and writes were proxied along the same tree traversal chain that would be followed on a read miss. RYW would be violated if any of the following were true: user requests were routed to another cluster of web servers; the mapping from web server cluster to L1 cache cluster was changed; queries were failed over to a stale replica; cache contents were lost before asynchronous replication occurred; or any query was served by a data store other than TAO.

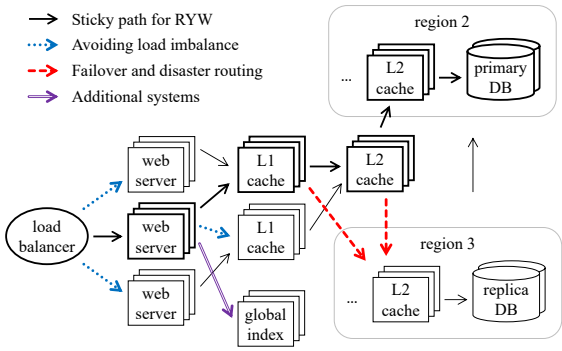


Figure 2: RYW via write-through caching excludes many useful inter-cluster communication patterns.

2.1 Scaling challenges

As TAO's footprint grew, we found it increasingly problematic to rely on a fixed communication topology. In fact, each of the conditions required for write-through RYW became harder to satisfy over time. As cross-cluster networking improved, we moved away from pairing and collocating L1 caches with web server clusters, reducing the number of L1 cache replicas per region. This reduced the number of cached copies of data, but it required fractional or dynamic assignment of web server traffic to L1 cache clusters to get reasonable balance. Switching from cluster-sticky to region-sticky user routing improved the load distribution of both the web server clusters and TAO. As the number of geographic regions grew, we started to deploy TAO in some datacenters without a local database replica, routing cache misses to the closest neighboring region. If we were restricted to a tree topology for miss routing and cache invalidation streams, the outage of one database replica would affect multiple regions. Figure 2 shows some of the desirable communication patterns we encountered that break the write-through consistency model. The dotted and dashed arrows show read requests that potentially violate RYW consistency without FlightTracker.

Another recurring issue was queries that needed cross-cluster or global write visibility. TAO marks these queries *critical*, routing them to the L2 cache in the region holding database primaries, near the base of the communication tree [20]. This strategy has latency and availability drawbacks. It is also not tolerant of spiky workloads.

2.2 Cross-system consistency

As we encountered challenges scaling TAO's write-through approach to consistency, the social graph ecosystem expanded. Application developers moved from directly accessing TAO to using a query language that makes it easy to express multi-hop and attribute-filtering predicates over the graph. This layer of indirection allowed us to build additional systems tailored to a subset of the Facebook query workload.

Some application queries involve many round trips when mapped onto TAO's simple API and transfer a lot of data that the client immediately discards. Global secondary indexes can optimize the communication pattern of these queries, but it is only safe to transparently or retroactively optimize execution using indexes if the index stores have the same semantics as TAO [34]. Our indexing systems are loosely coupled, updated by asynchronous pipelines that reshard, transform, and filter. Loose coupling enables separate development and deployment, but it limits the consistency implementation strategies. Most indexes are sharded differently than TAO, so even if we used a more monolithic design, they could not participate synchronously in the write path without reducing availability and increasing tail latency [17, 54].

Another side effect of moving to the application-level query language for the social graph was that it became easier to use alternate database technologies as the system of record for parts of the social graph, such as for data types that experience high write rates or limited lifetimes. These systems also experience the same consistency challenges as TAO.

Ajoux et al. [10] previously identified four fundamental challenges to providing causal consistency in Facebook's social media platform: integrating across many stateful services, tolerating high query amplification, handling linchpin objects (i.e., hot spots), and providing a net benefit for users. Our experience has been that these challenges also arise when providing RYW consistency and that the most difficult hurdle is producing a design that addresses all of them simultaneously.

2.3 Why read-your-writes?

Consistency models for data stores make guarantees about what writes should be visible to a read. Application developers use these guarantees to reason about the correctness of the entire system. Strong models like linearizability [32] or causal consistency [9] generally provide a simpler experience and mental model for developers, but they constrain the implementation.

Providing consistency guarantees for read-optimized systems boils down to implementing a staleness check to determine whether a cache or replica can serve a read query with its local data. This staleness check must be: (1) local, avoiding network communication in most cases; (2) highly granular, so that few queries result in extra work due to false positives from the checks; and (3) conducive to incremental repair, so that the extra work to find fresh data can be reused for subsequent queries. Importantly, staleness checks are needed for single-replica reads even in systems that use synchronous quorum writes. For example, Raft followers [43] or Paxos acceptors [38] might have no knowledge of a write committed by the leader if they were not part of the commit quorum.

Logical and physical timestamps, such as Hybrid logical clocks [36], Spanner's TrueTime [26], and Occult's compressed vector clocks [41], provide a simple and scalable way to check for staleness—the local data is sufficiently fresh if its timestamp is higher than the desired read timestamp. Unfortunately, these approaches are neither granular nor conducive to incremental repair. If the local store is 10 seconds behind the desired read timestamp, for example, it cannot service any queries until it has processed all of the missing writes.

For our workload, it is important that we can serve most queries locally, even if the local replicas are a few seconds stale. This led us to reject consistency levels in which all writes (linearizability) or most writes (causal consistency) missing from a stale replica need to be visible. In contrast, RYW allows us to utilize a stale replica by adding fresh versions of only a limited set of writes ("your" writes). We also rejected weaker models like bounded staleness that do not guarantee that a user sees their own writes, which are difficult to use correctly for an interactive application [54].[1]

Our experience at Facebook has been that the simple RYW consistency model [51] is a reasonable default for application developers and our end users, with an extension: we want to extend the concept of a session to end users.

User-centric sessions: Our desire to implement user-centric session RYW guarantees means that we experience intra-session concurrency at several levels, as shown in Figure 1: a single web request issues TAO reads and writes in parallel; a single browser or mobile app has many web requests in flight at once; and a user may even be accessing Facebook simultaneously from multiple devices.

The original definition of RYW session guarantees [51] implies that reads and writes within a session are totally ordered and that this intra-session order coincides with the physical order of those operations, as in linearizability. As a result, while the theoretical definition does not require a single-threaded session, implementations limit sessions to a single client, writer, session manager, or server [19, 28, 42, 52].

However, our observation is that application developers do not expect concurrent web requests to communicate with each other. A common mental model is that concurrent requests execute in a random order and possibly interleave with each other, so visibility from one request to the next is only assured if one request finishes before the other starts. An application developer's intuition is that the first moment data is guaranteed to be available is when the write acknowledgement is received by the local program that issued the web request.

This observation led us to the following relaxation of the RYW guarantee, which is what FlightTracker provides:

A read to the social graph will observe all writes done by the same end user in previously completed web requests or in the same web request.

This definition gives us much-needed flexibility for handling intra-session concurrency. Note that as in the original RYW definition, a read may "observe" a write by returning an even newer version of the data.

[1] We think providing both RYW and bounded staleness is an interesting and feasible model, even for our read-optimized environment.

3 FlightTracker

The main idea of FlightTracker is to decompose the problem of RYW consistency into three parts: (1) the Ticket abstraction, a flexible and extensible way of representing write sets across independently developed systems and APIs; (2) the FlightTracker service, generic infrastructure queried once per web request to get a user's recent write metadata; and (3) Ticket-inclusive reads, system-specific mechanisms to ensure that the specified writes are reflected in query results.

Our goal for FlightTracker is to preserve the communication patterns that benefit eventually consistent read-optimized stores. FlightTracker piggybacks on existing messages in these data stores. Most read queries can be served by a single local RPC, maintaining high efficiency and low latency. FlightTracker does not restrict where data stores send read RPCs, which allows them to leverage per-query retry and failover for high availability. FlightTracker supports the aggressive multi-level caching that data stores use to tolerate hot spots. Most of the work to ensure writes become visible to reads is handled by asynchronous pipelines, which retains the desired isolation and loose coupling of the underlying data stores. This piggybacking approach has also made it feasible to incrementally add FlightTracker support to existing mature systems with low overhead.

Figure 3 shows the API extensions a data store needs to implement to integrate with FlightTracker. On a successful write, a data store returns a Ticket identifying the write alongside the result; read queries take a Ticket parameter and guarantee any relevant writes in the Ticket will be reflected in the result.

3.1 An example

Consider a hypothetical social media product using TAO's graph model of versioned nodes and edges, with user nodes, media nodes, edges when a user has enjoyed a particular media instance, and edges when a user trusts another's tastes. Let's say Alice enjoys Mozart's *Requiem*; Bob recently indicated he trusts Alice's taste in art and then expanded his trust in Alice to include music. The resulting subgraph is shown in Figure 5 and Bob's recent writes to TAO would be:

$$\mathit{WriteEdge}(\langle 17, \texttt{TRUSTS}, 42\rangle \mapsto \{\texttt{"art"}\}) \quad (1)$$
$$\mathit{WriteEdge}(\langle 42, \texttt{TRUSTED_BY}, 17\rangle \mapsto \{\texttt{"art"}\}) \quad (2)$$
$$\mathit{WriteEdge}(\langle 17, \texttt{TRUSTS}, 42\rangle \mapsto \{\texttt{"art"}, \texttt{"music"}\}) \quad (3)$$
$$\mathit{WriteEdge}(\langle 42, \texttt{TRUSTED_BY}, 17\rangle \mapsto \{\texttt{"art"}, \texttt{"m...}\}) \quad (4)$$

To get RYW consistency, we simply need to ensure that Bob's subsequent data store queries include the effects of these writes. We do this by computing Bob's recent write set once per web request, attaching it to all of his queries, and then making sure the data stores reflect attached writes in the query results.

3.2 Tickets

We store write metadata in a data type we call a Ticket. Metadata to identify a write includes information like the transaction ID and the resulting node or edge version but does not include the data itself. If W_i is metadata that identifies Bob's i-th write above, we might have:

$$W_3 = [\mathit{key} \mapsto \langle 17, \texttt{TRUSTS}, 42\rangle, \\ \mathit{op} \mapsto \mathit{WriteEdge},\ v \mapsto 2,\ \mathit{txn_id} \mapsto 8980]$$
$$W_4 = [\mathit{key} \mapsto \langle 42, \texttt{TRUSTED_BY}, 17\rangle,\ \ldots,\ \mathit{txn_id} \mapsto 8985]$$

Bob's Ticket would then be $\{W_1 \ldots W_4\}$.

As write sets, Tickets can be joined via set union. Moreover, Tickets are handled and passed around between many independently deployed systems; therefore, they need to be encapsulated, extensible, and forward- and backward-compatible. Inside a Ticket, writes can be enumerated or represented using a low-water mark that implicitly includes all preceding writes. § 4 describes Ticket contents, semantics, and implementation.

```
pair<Result, Ticket> write(...); // returns metadata
Result read(..., Ticket); // Ticket-inclusive read
```

Figure 3: API extensions data stores expose to participate in the FlightTracker ecosystem.

```
void appendWrite(SessionId, Ticket); // FT write
Ticket getMergedWrites(SessionId); // FT read
```

Figure 4: The API of the FlightTracker service.

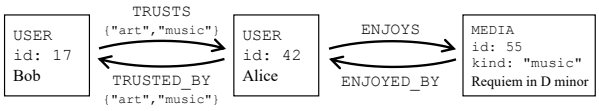


Figure 5: Subgraph for a hypothetical application.

3.3 The FlightTracker service

Bob's logical user session spans many web requests, so we need to store metadata for his recent writes elsewhere. To that end, we built the FlightTracker service with an API resembling a hash map of user IDs to recent writes (Figure 4). For example, Bob's entry will be $17 \mapsto \{W_1, W_2, W_3, W_4\}$. The client library calls `appendWrite` immediately after a successful write to the data store before acknowledging success to the application; `getMergedWrites` returns the recent writes for a particular user.

Figure 1 shows an RPC pattern that might occur as Bob browses the music portion of the site. As soon as the web request identifies Bob as the logged-in user, it fetches his RYW Ticket from FlightTracker by calling `getMergedWrites(17)` and puts it into the web request context. When Bob performs a write, the client library joins its metadata into the Ticket in the web request context and also uses `appendWrite` to immediately send the write metadata to FlightTracker. The client library implicitly attaches the Ticket from the web request context to every read query. A single web request performs many such queries, offering ample opportunity to amortize the initial Ticket fetch. Most developers do not explicitly observe or manipulate Tickets.

3.4 Ticket-inclusive reads

It is the responsibility of the data store clients to attach a Ticket that ensures RYW to each query, and it is the responsibility of each query-serving component to ensure that all of the writes in the Ticket are included in a query result.

Our applications do not expect to have exclusive access to the social graph or to read from snapshots; reads are always allowed to return data that is fresher than expected. In Ticket-inclusive reads, a Ticket specifies a lower bound on writes that should be visible. A Ticket that encodes a superset of another can always be safely substituted at read time, as anything made visible by the superset Ticket might have been visible anyway as part of normal asynchronous replication.

Cache queries: After getting his RYW Ticket, Bob's web request performs two queries to TAO's cache. The simplicity of the TAO API makes it straightforward for the cache to validate the freshness of its cache content—a TAO replica compares the versions of the data in question against the versions specified in the Ticket. For example, if the request is reading a list of all of the users that Bob trusts ($GetEdges(\langle 17, \texttt{TRUSTS}, * \rangle)$), then W_3 implies the edge to Alice must be present with version ≥ 2.

In Figure 1, the first TAO query was a cache hit unaffected by the Ticket. This is the common case. The second TAO query shows a *consistency miss*, where the local cache contents are stale. In this case, the cache goes upstream and merges the fresh edge into the local list before responding. Note that the upstream query has the same Ticket attached, which recursively ensures visibility of Bob's recent writes.

Index queries: If Bob is browsing the song with ID 55, we would like to display Bob's trusted users who also enjoy it. This involves finding all x where $\langle 17, \texttt{TRUSTS["music"]}, x \rangle \wedge \langle 55, \texttt{ENJOYED_BY}, x \rangle$. This two-hop query is not well suited to TAO, because both the `TRUSTS` and `ENJOYED_BY` edge lists may be too large to fully cache. We can optimize this type of query by materializing a global secondary index. Specifically, we might use a list-intersection index with edge lists for `TRUSTS` edges that include `"music"` and `ENJOYED_BY` edges from `"music"` `MEDIA` nodes.

An index leaf (read server) does not have enough information to accurately identify missing writes, because writes that are filtered by the update pipeline will never arrive. For example, W_1 can be filtered upstream because it is not a `TRUSTS` edge that includes `"music"` and thus does not change any materialized lists. Without extra information, an index leaf will consider W_1 missing forever. Any such index leaf cannot satisfy a Ticket-inclusive read with Bob's Ticket with W_1 in it. We solve this problem by tracking the delivery information of recent writes, including the recent routing and filtering choices of the update pipeline as well as the delivery status to the index leaves. This FlightTracker-ReverseIndex (FT-RI) component builds an index of recent writes to the actions taken by the index update pipeline (§ 6.3). Queryable using the metadata present in Tickets, the delivery information is used by the index client library to determine whether an index read result is fresh enough. If stale, the client library uses strategies such as read repair or retry to obtain a fresh result.

§ 6 describes our full range of strategies to ensure results of Ticket-inclusive reads reflect all writes in a Ticket.

4 Ticket details

A Ticket is a set of write metadata. We use Tickets to identify writes to the social graph regardless of where the writes are committed. Tickets allow generic infrastructure to track and identify writes across many independently deployed systems, while letting databases convey system-specific details. For clarity, we refer to the systems that persist the normalized data and generate the write metadata as "databases," as opposed to other data stores such as caches and indexes which mostly serve reads and proxy writes.

A Ticket is implemented as a union of custom per-database representations. On a write, a single-database Ticket is minted with only metadata for the newly committed write (Figure 3). It can then be joined with other Tickets or otherwise used by FlightTracker and custom applications, producing Tickets that may contain writes from multiple databases.

The encapsulation of Tickets and the semantics of Ticket-inclusive reads together give us great flexibility in the Ticket implementation. Since Ticket-inclusive reads interpret Tickets as lower bounds, read results containing additional writes or fresher writes than exactly encoded in the Ticket are unsurprising to the applications. Furthermore, thanks to encapsulation, applications cannot examine the exact content or representation of a Ticket, which means we can always safely include additional writes inside a Ticket. We leverage this flexibility in Ticket compaction (§ 4.2) and Ticket replication in the FlightTracker service (§ 5).

4.1 Identifying a set of writes

The naive strategy of identifying writes by globally unique IDs is easy to implement but difficult to use—read-serving data stores must keep track of all write IDs to determine whether the local replica is sufficiently up-to-date. Assigning a total write order allows systems to identify writes by their ordinal positions. However, ordering implies synchronization via communication or via timed-wait [26]. To preserve the efficiency benefits of asynchronous replication, databases often opt for limited-scope ordering. In our experiences, there are three natural scopes:

- Per-key: Any strictly monotonically increasing value can be combined with a key to identify a particular change to that row or object. This version need not be contiguous—a monotonically increasing timestamp would also suffice.
- Per-shard: Many databases totally order all writes to a particular shard, but give no order guarantees between shards.

- Global: Some systems [26,53] offer total order of all writes globally. HLCs or known-accuracy clocks can also order writes across shards and database types.

All of the databases supported by FlightTracker have the following properties which allow us to further simplify our assumptions. Our databases expose a versioned key-object model. They are sharded and maintain a total order for all writes within a shard; furthermore, writes in a single shard are replicated in the same order. The commit-time information (such as commit timestamps or transaction IDs) can thus specify a contiguous prefix of that shard and serve as a low-water mark to determine the replication state of a data store.

Most fields in a Ticket can be and are empty. When new Tickets are minted, they can include any information known at commit time, such as per-key versions, commit timestamps, or transaction IDs. While not necessary, including more metadata allows flexibility in interpreting and using the Ticket. Figure 6 shows an example Ticket structure with two databases.

For a data store or client to determine whether a read result is sufficiently up-to-date, this metadata needs to be accessible on reads, which means that it should be stored alongside the data. While this overhead appears non-trivial (e.g., up to 8-bytes per key), our databases already persist these versioning primitives, so the additional cost is negligible.

Although we describe the Ticket abstraction in the context of databases at Facebook, it is applicable and extensible to other databases. Many natively support and store per-key versioning primitives, such as the `rowversion` of Azure SQL Database [13]. Per-shard or global-scope ordering also often underlies modern databases, where writes can be identified by sequence numbers or timestamps stored alongside the client data (e.g., `zxid` for ZooKeeper [33], `hlc` in CockroachDB [2], `offset` in Kafka [35], `LSN` in LogDevice [4]).

4.2 Ticket joining and compaction

Two important operations on Tickets are joining and compaction: joining combines Tickets and compaction reduces their space footprint. Both are local operations with no need for RPCs or information outside the input Tickets.

The *join* operation, which is essentially set union, produces a Ticket that is a superset of all the inputs. It is the primary API for Tickets. For example, data store clients can join Tickets to combine metadata from multiple shards or multiple databases; the FlightTracker service joins Tickets to accumulate per-user recent writes (§ 5); select applications join Tickets to express additional constraints for their reads (§ 7).

Compaction helps Tickets overcome the scaling limit of write set tracking techniques. The idea is straightforward. Tickets represent writes that should be visible, i.e., a kind of lower bound; we can raise the lower bound in exchange for a more compact representation. Intuitively, doing Ticket-inclusive reads with the resulting Ticket makes their constraints equally or more stringent.

Formally, Tickets are CRDTs [50] and the compaction techniques we use are CRDT *inflation* operations [18]: the per-scope ordering and subset-superset relation define the $\leq$ partial order. Ticket inflation produces a Ticket that is $\geq$ the input Ticket according to this order. The resulting Ticket may need fewer bytes to represent. Not all inflation reduces Ticket size but the three types of inflation we use below do:

```
struct RepForDatabaseA {
  map<WriteKey, tuple<Version, TxnId, Timestamp>> perKeyMap;
  map<ShardId, pair<TxnId, Timestamp>> perShardMap;
};
struct RepForDatabaseB {
  map<WriteKey, tuple<Version, Hlc>> perKeyMap;
  map<ShardId, Hlc> perShardMap;
};
struct Ticket {
  RepForDatabaseA repA;
  RepForDatabaseB repB;
  Timestamp globalTs;
};
```

Figure 6: Tickets represent the union of the writes identified by each field.

Per-scope compaction: Keeping the highest version for each key and the highest transaction ID for each shard lets us discard metadata with older versions or transaction IDs. This compaction is performed during every join.

Cross-scope compaction: Some databases have static shard assignments and include both per-key versions and per-shard transaction IDs in the Tickets (such as `DatabaseA` in Figure 6). Replacing per-key metadata with a per-shard transaction ID can greatly reduce the Ticket size, especially for shards with many individual writes. Suppose the edges from the example in § 3.1 among many other writes are all on shard X. We can then compact a Ticket $T_1 = \{W_3, W_4, \ldots, W_{100}\}$ to $T_2 = \{shard_X : \ txn_id \mapsto 8985\}$.

Cross-scope compaction offers us a tradeoff between the Ticket size and the cost of serving the Ticket-inclusive read. Since a compacted Ticket semantically encodes more writes, the read is less likely to be served locally. E.g., T_2 requires all writes on shard X with $txn_id \leq 8985$ to be replicated. Thus, this type of compaction is performed heuristically and sparingly in the FlightTracker service.

Global compaction: We can inflate a Ticket into a global-scope timestamp to represent all writes that have earlier timestamps. Global compaction only happens in the FlightTracker service for writes older than 60 seconds, since we assume the replication lag of our data stores plus clock skews are much smaller. Given the long threshold, global compaction does not have to be exact—older writes do not need to be removed from a Ticket immediately since they purely reduce Ticket size. Thus, the timestamps used for compaction could either be the logical commit timestamp generated by the database or the physical timestamp generated by the data store after the write completes. Global compaction lets FlightTracker store only 60s of data, which greatly reduces its working set.

Additionally, we define a Ticket-inclusive read with an

empty Ticket as implicitly encoding the constraint of returning data no more than 60 seconds stale relative to the current physical timestamp of the replica serving the read. This way, in most cases, we avoid the need to pass around Tickets containing only a single old global timestamp.

4.3 Physical representation

As a cross-system primitive, Ticket presents a number of interesting software-engineering challenges. Ticket-handling code runs inside systems with widely varying deployment frequencies, so Ticket must be both forward- and backward-**compatible**. Ticket **encapsulates** implementation-specific details from multiple systems, but because clients can join Tickets, it cannot leave encoding and decoding up to the data stores. Ticket must be **extensible** and **loosely coupled**, allowing metadata for new systems to be added without affecting existing systems. Ticket must also be **efficient** enough to use on each query in our read-heavy environment.

We address some of the above challenges by using Thrift [8], a serialization format originally designed for efficient and portable RPC. We define the Ticket data structure using the Thrift interface definition language (IDL).

Compatibility: Thrift handles the majority of forward- and backward-compatibility issues, as unknown fields from future versions are silently skipped. Care must still be taken as we introduce new metadata fields to existing systems in the Ticket. For example, if two writes with the same key and timestamp are differentiated by a transaction ID, older code unaware of transaction IDs may be surprised to see a duplicated write.

Serialization: We currently support two serialization formats, identified by the prefix. The default is an LZ4-compression [25] of the Thrift Compact encoding. This is used across all RPC boundaries, making it easy to tunnel Tickets through other systems. The second is the Thrift JSON encoding for readability in debugging and logging.

Encapsulation: A Ticket's internal struct is accessible to systems that mint new Tickets or perform Ticket-inclusive reads. Clients that need not inspect Tickets can treat them as opaque tokens. We provide language bindings and utility functions for code that needs to examine the Ticket internals.

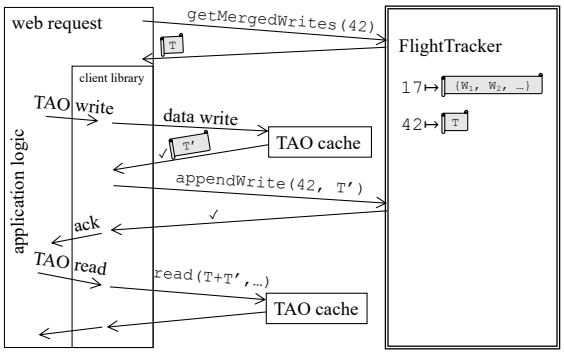


Figure 7: Web request flow with the FlightTracker service.

We chose the name *Ticket* to minimize assumptions developers would make about its semantics. Infrastructure engineers often conflate a visibility guarantee on a specific write (Transaction 8980) with a guarantee on a contiguous prefix (Transactions 1…8980); a new name reduces this tendency.

Extensibility and loose coupling: As shown in Figure 6, each database can customize its own representation. Extending the Ticket structure to support an additional database boils down to adding a field in the main struct and updating the join function. Loose coupling between databases is provided by using different fields for each.

5 FlightTracker service implementation

The API of the stateful FlightTracker service (Figure 4) is extremely simple, consisting of just two operations. As shown in Figure 7, a web request calls `getMergedWrites(user)` at the beginning to get the user session's RYW Ticket; the client library call `appendWrite(user, ticket)` after a database write with the newly minted Ticket and only acknowledge the write to the application if *both* the data store write and `appendWrite` succeed. To reduce ambiguity, we use "data writes" to refer to data store operations and use "metadata writes" to refer to FlightTracker operations.

FlightTracker has the following requirements:

- High throughput: FlightTracker is subject to the full write throughput of all the underlying data stores, since every data write results in a metadata write to FlightTracker. Its effective replication factor is lower than a globally replicated store like TAO, because most writes are only stored in the writing user's region. Its read throughput is proportional to the number of web requests.
- Low latency: Data writes are not acknowledged until their metadata is recorded in FlightTracker, so FlightTracker adds to application-visible write latency.
- High availability: Unavailability of FlightTracker implies loss of availability or loss of RYW consistency for clients. The decoupled nature of FlightTracker allows us to let some use cases fail open (available but inconsistent) while others fail closed (unavailable).
- Durability: A Ticket passed to `appendWrite` should be included in `getMergedWrites` even when there are machine failures. FlightTracker uses a single-round quorum protocol that does not provide atomicity because it is okay for a failed or in-progress `appendWrite` call to be visible.

The working set of FlightTracker is relatively small, as FlightTracker compacts Tickets as it merges them (§ 4.2). Put more practically, if queries are only routed to replicas that are at most 60s stale, then write metadata older than 60s are safe to compact away. Our data stores track the staleness of their own replication streams and a vast majority (>99.99%) of the servers are no more than 60s stale.

5.1 Replication

We implement FlightTracker as a quorum-based store. We statically determine the number of replicas N; the FlightTracker client broadcasts an `appendWrite` to all replicas and considers it successful when W replicas acknowledge the write; a `getMergedWrites` query contacts R replicas and joins the retrieved Tickets.

This plain *single-round* quorum protocol with $R+W>N$ is sufficient to provide the desired correctness guarantee for FlightTracker. A previously appended Ticket will be returned by at least one replica in R, since $R+W>N$ guarantees overlap between the read and write quorums. Merging the read results via `join()` from all R replicas ensures durability: the Ticket will be included in the final read result.

FlightTracker does not need to guarantee atomicity. Recall that given Ticket encapsulation and how Tickets are used as lower bounds, we can safely include additional write metadata in Tickets without violating the overall RYW consistency (§ 4). If a metadata write fails to reach W FlightTracker replicas or is still in progress, FlightTracker *can* safely include it in the result of `getMergedWrites`. Moreover, if a data write succeeds but its metadata write to FlightTracker fails, we consider this write an "unacknowledged success," i.e., the data store client errors out the data write to the application. Application developers do not expect to see failed data writes but know how to handle them if they do show up. Since many of Facebook's applications are built on eventually consistent stores, applications are used to reading fresher writes (e.g., from other applications). Thus, unacknowledged successes are acceptable as long as they are infrequent.

For example, suppose we have a FlightTracker deployment with $N=W=3$ and $R=1$; data writes for W_5 and W_6 completed, but the metadata write for W_5 failed and the metadata write for W_6 is in progress. The state of the three FlightTracker replicas is $\{W_5,W_6\},\{W_6\},\{\}$. A first metadata read could return a Ticket of $\{W_5,W_6\}$ and a subsequent metadata read could return $\{\}$. This is permitted: W_6 has not completed so RYW does not apply yet; W_5, while written in the database, has returned an error to the application, thus the application should have no expectation of visibility either way.

FlightTracker's default setup is a region-local quorum, as Facebook pins logged-in users to a region. We leverage the regional placement to ensure low latency for FlightTracker accesses. Since FlightTracker has a small working set, we choose to store everything in memory and adjust N for redundancy. Empirically, we've found $N=3$ offers a good trade-off between low latency and sufficient redundancy (§ 8).

We statically map user IDs onto logical shards, which are dynamically placed within each FlightTracker replica. Shard placement is aware of load-balancing and covers failure detection. Some use cases use non-user session IDs and cross-region quorums (§ 7.3).

5.2 Failure tolerance

Machine failure is the most common failure that must be handled. Our strategy for this also handles network issues and gaps in coverage during shard movements. We leverage the global compaction bound to restore resiliency after a FlightTracker machine gets a new shard assignment or dynamic shard movement. FlightTracker "warms up" in the first 60 seconds after a shard comes online by accepting all writes but not serving reads. Rejected reads are retried on warm replicas.

5.3 Fail closed vs. fail open

One of the challenges identified by Ajoux et al. [10] was ensuring consistency mechanisms provide a net user benefit. Some applications would prefer to continue a web request even if `getMergedWrites` fails, for example. Given that we have the option to cleanly fail open on a per-query basis, it is difficult to argue for a uniform fail-closed policy. If FlightTracker is completely reliable, there will be vanishingly few inconsistencies even with a fail-open policy, so there is no benefit to fail-closed; on the other hand, if FlightTracker is not completely reliable, then use cases that prefer availability will be harmed by fail-closed. A future option would be to rate-limit fail-closed for those use cases and escalate all fail-open potential RYW violations to an engineer.

Since an error is reported to the application when a data write succeeds but the corresponding `appendWrite` fails, FlightTracker write availability caps the data store availability. FlightTracker is in-memory and region-local, so it has much higher write availability (§ 8) than our underlying persistent data stores; it has a minimal impact on application-visible write availability. Although less important now, the option to fail open was crucial to reducing risk during early rollout.

6 Ticket-inclusive reads

Once FlightTracker has attached a Ticket to a query, it is the responsibility of the data store to ensure that every write identified by the Ticket is reflected in the query result.

The general pattern for implementing Ticket-inclusive reads is that the data store (client or server) filters the writes in the Ticket for relevancy, then checks against its local state to see if they have already been applied. Frequently, the writes in a Ticket are irrelevant to the read (e.g., a write to a `MEDIA` node is irrelevant to reading Alice's `TRUSTED` users) or have been replicated and included locally, in which case the Ticket does not change the result and the read can be served locally.

In the uncommon case that some writes are possibly relevant but missing, i.e., the local data is *possibly* stale, the data store uses a more expensive non-local action to fix the query result. The specific strategies for each of those steps depend on query semantics, the way in which writes are encoded in the Ticket, and what information is locally available.

6.1 Filtering by relevance

Filtering works best for the most granular write representations such as per-key versions. In contrast, timestamps define a write set that includes a contiguous prefix of the history of all data systems at Facebook, which can never be filtered.

For database-specific encodings, a coarse level of filtering happens when the data store ignores writes from other databases in the Ticket. It is also fast and easy to filter by static information in the Ticket or in data store configs, such as TAO object types (e.g., `USER` or `ENJOYS`) or database tables. For Ticket representations that contain keys, we can further filter writes based on query parameters such as the desired node ID. This is highly effective for point queries and simple range queries like TAO's and works for some indexes.

Type and query parameter filtering can be done on the client, which avoids the need to even include a Ticket on most queries. We refer to this operation as *cropping* and have integrated it in the client libraries of all data stores we support.

6.2 Checking inclusion

The systems that incorporate FlightTracker provide eventual consistency on their own, mostly using asynchronous replication. The large majority of writes are delivered with low latency, so most writes are included at the check time.

For database replicas and caches sharded by key, we replicate in write order (i.e., in per-shard *txn_id* order). The replication stream pointer is a compact way of identifying the set of writes included in the local store. If the latest replicated record was 8983, for example, then the write W_3 with $txn_id \mapsto 8980$ is included but W_4 with $txn_id \mapsto 8985$ is not.

Cache misses can fetch values from ahead of the replication pointer, so a single low-water mark is not sufficient for high-granularity inclusion checks. TAO maintains a $key \rightarrow txn_id_{safe}$ mapping that identifies when a particular cache entry is known to include writes newer than the local low-water mark. txn_id_{safe} records the replication position of the upstream source when it serviced the cache miss, not necessarily the transaction at which the key was updated. For example, if a cache with low-water mark 8983 took a cache miss and fetched the edge in W_4 from a database replicated up to 9000, it will have a cache entry with $txn_id_{safe} \mapsto 9000$ for the edge; the cache is now able to serve point reads to the edge locally if the Ticket has W_4 or even $\{shard_X : txn_id \mapsto 9000\}$. This exception map is essential to ensuring that Ticket-inclusive queries can still be cache hits.

6.3 Relevance and inclusion for global indexes

Both relevance and inclusion checking are much more challenging for global indexes. An index that lets us find media nodes by their name, for example, will be partitioned using a mutable data attribute rather than by the node's key.

While in this case it would be feasible to include the indexed attribute in the Ticket, we avoid this approach. It bloats the Ticket without solving the problem for all indexes, because the indexed attributes might be from adjacent nodes or edges in the graph. It requires writers to be aware of all index schemas and cannot scale to handle fan-in cases, where a single write affects a large number of index rows. In the example in § 3.4 where we want to answer queries such as "return a list of trusted users who also enjoy a particular song," the graph indexing system materializes an ordered list of $\langle user.id,\ media.id,\ list(trusted_user.id)\rangle$ tuples. Checking whether a write like W_3 (which expands the `TRUSTS` edge to include "music" between Bob and Alice) is relevant or locally applied to an index server requires the list of `MEDIA` nodes that Alice `ENJOYS`. This list could easily bloat the Ticket should we take this approach.

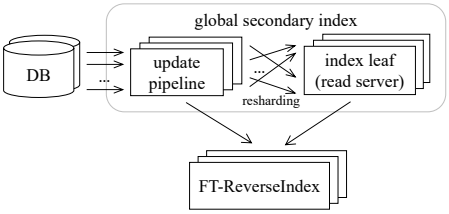


Figure 8: Stages of the asynchronous index update pipeline inform FT-RI as they process writes.

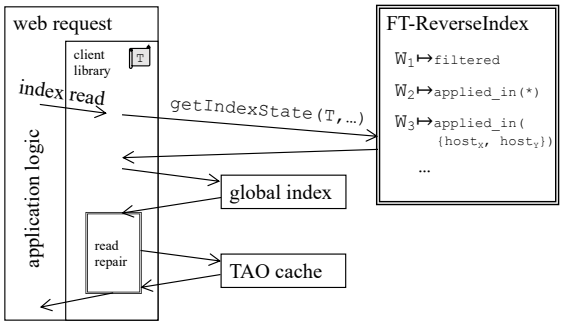


Figure 9: Web request flow with FT-ReverseIndex.

Another option we rejected was to ignore relevance checking for indexes and focus only on inclusion. This would require plumbing information about the replication water marks of all shards through the index update pipeline, perhaps using a compressed vector clock scheme like Occult [41] to avoid the need to deterministically merge across millions of replication streams. To distinguish lack of new updates from staleness in replication, each stream needs heartbeats, which results in a lot of overhead for cold shards and small indexes.

Our solution is to build an inverted index from writes to the actions taken by the index update pipeline. We store this in a stateful component named *FlightTracker-ReverseIndex* (FT-RI). We describe the interactions between the index update pipeline and FT-RI shown in Figure 8 with the example of W_3 and the intersection index. Based on W_3's type and index schemas, FT-RI determines the indexes W_3 could affect and initially assumes it could affect every row in those indexes. As W_3 goes through the update pipeline, each stage of the pipeline informs FT-RI when it is about to filter out W_3 for some or

all indexes. We also require the update pipeline to propagate W_3's metadata all the way to the index leaf servers unless W_3 is filtered out entirely. The update pipeline determines that W_3 matters only for index rows of the form $\langle 17,\ media.id,\ 42\rangle$ in the intersection index and informs FT-RI. When a leaf applies the index row updates generated due to W_3, it informs FT-RI of its server identifier and the part of the index state was updated. This way, FT-RI narrows down the scope of the indexes and read queries W_3 might affect.

As shown in Figure 9, to perform both relevance filtering and inclusion checking for a Ticket-inclusive index read, the client library first sends the query and the Ticket to FT-RI. FT-RI then returns the subset of writes that might be relevant (since they have not been reported as filtered) and not yet included (since they are still missing from the index leaves).

The client consults FT-RI before the query is sent to the index, so the set of missing writes may include false positives. False negatives would lead to RYW violations, so they must be avoided. To minimize the false positive rate without introducing any false negatives, FT-RI returns a map from writes to the physical servers where it may be missing. The client checks this information against the query execution plan, in case the stale server was not actually consulted. It can also be used to make intelligent replica choices and to retry only the stale portion of a query.

FT-RI accumulates a set of irrefutable facts about writes, so its internal state is a CRDT. It exploits the same single-round quorum protocol as FlightTracker (§ 5) for replication. FT-RI also shares much of the same infrastructure and is deployed as a RAM-only regional service.

6.4 Strategies to handle local staleness

This section describes ways to get the correct result when a potentially relevant write might be missing from the local data store. Our approaches fall into two categories: when the Ticket enumerates individual writes, the data store can request the data from upstream and cache the result for the next reader; if the Ticket contains a contiguous prefix, such as after compaction, we generally only reevaluate the query (on a different replica or at a later time), as it is expensive or impossible to request the contiguous prefix. In production, we use every strategy below but index repair.

Delay and retry: When we realize a data store is stale, a simple option is to just try again later. This strategy is not sufficient on its own, but it can be used as a first try to reduce the frequency of a more costly strategy.

Replica selection: Data stores are replicated for read availability. When one replica is stale, we can contact another replica that is up-to-date, especially if it is nearby. This strategy can lead to correlated failures such as thundering herd, so we only use it for low-volume workloads or behind a cache.

Consistency miss: When a Ticket identifies individual writes by key, caches that keep per-key versions (§ 4.1) can easily determine which data items are stale. They can use their normal miss-handling logic to pull data about the missing writes from their upstream source, passing on the Ticket to recursively ensure visibility.

Even client-side hot object caches can similarly take consistency misses, which otherwise rely on TTLs to get fresh data. This is integral to our tolerance of read hot spots (§ 6.5).

Index bypass (re-materialization): Indexing systems have the option to fall back to the source of normalized data to answer a read query, though this is an expensive option. Quite a few of our indexes are materialized on-demand, so this fallback functionality is already regularly exercised.

Read repair: Read repair looks for possible matches to components of an index predicate among the writes in a Ticket, uses point queries to a non-index store like TAO to evaluate the full predicate, and then fixes the index result accordingly. Read repair can reduce complexity and latency. Consider the example in § 3.4 where we want to find Bob's trusted users who also enjoy Song 55. If W_3 on edge $\langle 17, \texttt{TRUSTS}, 42\rangle$ is in the Ticket and the read repair library sees that node 42 (Alice) `ENJOYS` Song 55 from TAO, it adds node 42 into the result set.

Index read repair does not completely avoid extra communication on following queries, like consistency misses in a cache, but it avoids the need for future cross-region calls.

FT-RI filters out writes we do not need to repair. As shown in Figure 9, the client library queries FT-RI to find which of Bob's relevant writes have not yet been applied before querying the index and read repair. We initially tested read repair without FT-RI, treating every write a user had made in the last 60 seconds as undetermined. FT-RI for list intersection indexes has only a modest effect on the average number of edges to be checked for read repair, but it provides a dramatic reduction for the worst cases.

Read repair has its limitations. Firstly, certain indexes with aggregation cannot be read repaired. For example, if an index query only returns the size of the intersection, the read repair library would not know which writes have been applied in the result. Fortunately, the vast majority of our online index usage returns set results that can be repaired. Secondly, client-side read repair for complex indexes, such as ones that require traversing 3+ hops in the graph or those with large fan-outs, could duplicate the transform and processing logic of the update pipeline and index leaves, resulting in extra complexity. The above challenges are akin to those encountered in deferred incremental view maintenance in the database community [23, 24, 47, 58].

Index repair: Repairing the index by synchronously invoking the index update logic is more complex, but avoids many of the limitations of client-side read repair. We have not yet explored this option.

6.5 Handling hot spots

Handling linchpin objects is one of the major challenges of a social networking workload [10]. Read hot spots are a much

bigger concern than write hot spots in our read-heavy workload. Caches like TAO handle read hot spots by storing more local copies of the data, including on the client-side. Ticket-inclusive reads for cache queries are cache-able, preserving this hot spot tolerance. We cannot always cache post-repair index results, but the data fetched to perform repair is always locally cacheable.

Aside from hot objects for the overall system, FlightTracker has its own hot spots: since FlightTracker is sharded by `session_id`, it has different hot spot patterns from the underlying data stores. These hot spots are more likely due to user-triggered actions such as batch processing or from custom sessions (§ 7.3). To alleviate write hot spots, FlightTracker's client library batches metadata writes without concerns for sacrificing write availability, since all FlightTracker writes are conflict-free. On the FlightTracker server side, we proactively detect sessions that are frequently accessed. For a hot session that spawns many web requests and thus results in many metadata reads, we coalesce these metadata reads into short time buckets, and respond to all reads in a bucket with the same response. These strategies have eliminated hot spots as a significant error source for FlightTracker (§ 8).

7 Beyond RYW: Explicit write visibility

Certain applications need visibility guarantees beyond a single end user or across regions, where our default user-centric RYW consistency falls short. To obtain desired visibility guarantees, we enable them to explicitly manipulate Tickets or customize FlightTracker session IDs. These applications are responsible for explicitly identifying the writes and preventing the Tickets from growing too large.

7.1 Embedding Tickets in notifications

Facebook's notification infrastructure for GraphQL Subscriptions [48] fans out to all subscribers when a publisher event occurs. To render personalized notifications for each subscriber, GraphQL queries TAO in the subscriber regions. This pub-sub system races with TAO replication. To ensure that the query sees all of the writes associated with the event, we include and pass along the original publisher's Ticket. When rendering the notification, GraphQL transiently joins it with the subscriber's Ticket to query TAO. This is all hidden in the product infrastructure layer from product developers.

Subscriptions with a high subscribers fanout could result in a storm of consistency misses. Though TAO would only go cross-region once for this data, many requests would be stalled waiting for the result. Thus, we prefetch data in the Ticket into the local region's TAO before notification fanout.

7.2 Data-derived additional sessions

When a user performs a write that includes another person's User ID, such as when Bob created a `TRUSTS` edge to Alice, the write is naturally associated with the other user's FlightTracker session. For some edge types, we act on this by having the client library perform extra `appendWrites` calls, pushing the write to both the normal RYW session and the session identified by the destination node. These additional writes are sent to every region. We do not push the entire writing user's Ticket into the data-derived additional sessions; only the user-terminated edge write gets strengthened visibility guarantees. As users are highly connected [7], this conservative choice avoids the potential for super-linear growth of write sets.

7.3 Explicit global sessions

Some applications need visibility guarantees beyond a single end user or across region. We allow them to customize their session IDs and configure quorum and compaction in FlightTracker on a per-use-case basis.

Facebook's async job scheduling framework, similar to Celery or Resque [1, 5], enables web requests to schedule followup jobs such as sending email invites or long running migrations. These jobs may run in any region, but all of the writes from the original user session must be visible. To provide this guarantee, we use `job_id` as the session ID, which is the same for all tasks that are part of a job. Given the relative read-write ratio, we require a write to be replicated to most replicas in all regions and a read to be read from a few (usually region-local) replicas. We provide a utility function for the job framework to collect the writes the web request has done and send to FlightTracker under the appropriate job ID. When a job starts, it fetches a Ticket from FlightTracker using its job ID and uses Ticket-inclusive reads thereafter.

We also use global sessions for some TAO objects as an alternative to TAO's critical reads. Critical reads ensure write visibility by proxying reads to the region of the object's database primary, at the expense of increased latency, reduced efficiency, and reduced availability. If we record all writes to this object in a global session using its ID, we can replace the critical read: querying region-local FlightTracker to get a Ticket for this session and querying that object with a Ticket-inclusive read will return the latest successful (or newer) write. This approach shifts the cross-region latency to write time and increases read availability.

8 Evaluation

FlightTracker allows Facebook to get the read efficiency, hot spot tolerance, and high availability of eventual consistency while providing RYW consistency with a rich notion of user sessions that spans many stateful services.

8.1 Environment

Facebook serves millions of user requests per second. These user requests amplify to more than ten billion read queries per second to our online graph data stores, which also process tens of millions of writes per second.

Each of our stateful data stores, such as TAO and its indexes,

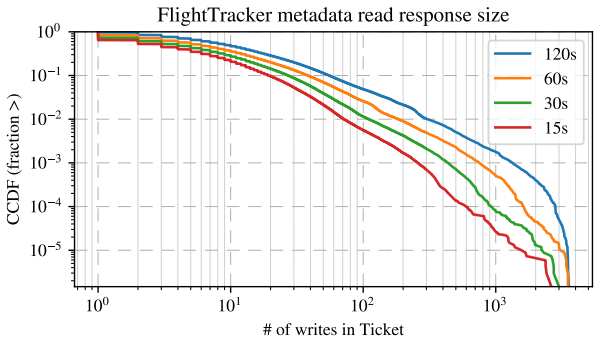


Figure 10: The size distribution of FlightTracker read responses in our production environment for different global compaction thresholds.

are deployed across over ten datacenter regions. Cache and indexing systems maintain asynchronously updated replicas in every region, while database replicas are present only in some regions. More than 99% of cache and index queries are served without any cross-region communication.

8.2 FlightTracker operational characteristics

FlightTracker has been in production for over four years. It manages RYW consistency for two database technologies, two cache types (including TAO), and three indexing systems. FlightTracker serves more than 100 million Ticket reads and 20 million Ticket writes per second.

We measured FlightTracker's availability as observed by the client over 30 days. Measuring errors from the client side offers an end-to-end picture, because it includes unavailability due to misconfiguration, networking issues, and collateral damage from other problems. FlightTracker's overall read error rate was 1.1×10^{-7}. When examining the availability data for 15-minute buckets, all but 8 buckets over the month of data indicated at least 99.9999% of read availability. FlightTracker's write availability was an order of magnitude higher than the write availability of the underlying databases.

8.3 FlightTracker overheads

Request and response sizes: The bane of explicit write-tracking techniques is handling large write sets. The Tickets fetched from FlightTracker contain all recent writes for a user that are not globally compacted (§ 4.2), so they tend to be the largest explicit write sets passed around in our systems. Figure 10 shows the size distribution of metadata read responses in production for different global compaction thresholds, measured in the number of writes in the returned Ticket. In production, we use 60s as the default, but as shown, extending it to 2 minutes does not significantly bend the curve.

Ticket serialization includes compression using LZ4 [25]. Figure 11 shows that this provides a useful benefit for Tickets with more individual writes, improving encoding efficiency by up to a factor of three. Table 1 shows that cropping in the client is effective; Ticket sizes attached to read queries are much smaller than the full write set pulled from FlightTracker.

Operation	Avg	P50	P99
FlightTracker metadata read response	250	0	2805
FlightTracker metadata write request	156	129	447
Ticket-inclusive read from cache	110	0	450
Ticket-inclusive read from indexes	225	152	607

Table 1: Serialized sizes of Tickets attached on various requests and responses, in bytes.

Operation	Avg	P50	P99
FlightTracker read	288 µs	226 µs	1.4 ms
FlightTracker write	376 µs	326 µs	1.5 ms
FT-ReverseIndex read	304 µs	236 µs	1.5 ms
FT-ReverseIndex update	428 µs	311 µs	1.2 ms

Table 2: Client-measured latency of FlightTracker and FT-RI.

Service	CPU	RAM
Application (web) servers	0.7%	0.06%
TAO L1 and L2 cache	0.8%	0.01%
Indexes and materialized views	0.98%	2.6%

Table 3: Relative CPU and memory costs of all code paths related to Ticket, FlightTracker, or FT-ReverseIndex.

Latency: FlightTracker and FT-RI have low latency for both reads and writes, as shown in Table 2. Both of these services are RAM-only and process all of their reads and writes from the local datacenter region. Queries for custom use cases (§ 7.3) are excluded in the table.

Footprint: The footprint of FlightTracker includes extra work and data in client libraries, extra work and space inside the data stores to enable Ticket-inclusive reads, and servers devoted exclusively to running the FlightTracker and FT-RI services. Table 3 shows that FlightTracker-related code paths consume only a small amount of the CPU and memory in clients and Ticket-enabled query-serving systems. The FlightTracker and FT-RI services use less than 2% as many servers as TAO and its indexes.

Extensibility: The Ticket abstraction is designed to be extended to handle new databases and new ways of encoding write metadata for the benefit of new projections. Since it was deployed to production, we have changed the Ticket Thrift schema 22 times, and we have made changes to the core Ticket join logic 50 times. Extending the Ticket abstraction to cover a new database does not increase the serialized size, but it does increase the heap footprint of the deserialized C++ objects. FlightTracker's RAM consumption increased by 5% when we added support for a second type of database.

8.4 FlightTracker effectiveness

RYW for caches: FlightTracker enabled our caches to no longer rely on fixed communication topology to provide RYW consistency. It provided an opportunity to apply additional techniques to improve efficiency and reliability for our caches.

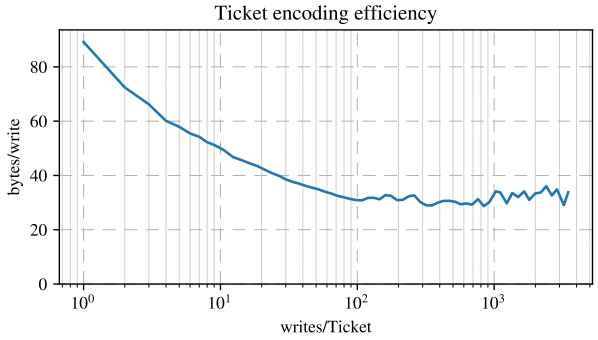


Figure 11: LZ4 improves serialized encoding efficiency by up to a factor of three for Tickets with more individual writes.

Today, 0.2% of the TAO reads have a non-empty Ticket attached, and 3% of those reads are for updates that have not yet been replicated via the per-shard replication stream. We allocate around 40MB per TAO instance for caching the result of Ticket-inclusive reads, resulting in a hit rate between 30% and 80% depending on the type of query. This hit rate is not evenly distributed: frequently read hot objects account for the bulk of hits. Fewer than 3% of TAO reads that end up going across regions are due to consistency misses.

Ticket-inclusive reads reduce cross-region traffic when they replace primary-DB-only queries for use cases that need stronger consistency guarantees. In one extreme use case where the cache only tracked per-shard replication progress, Ticket-inclusive reads reduced the percentage of queries going upstream from 20-40% to near 0%.

RYW for indexes: FlightTracker identifies that between 0.01% to 0.4% of indexing reads can benefit from read repair or other staleness handling strategies.

Explicit use cases: In the event delivery use case (§ 7.1), 3% of the subscribers' reads are for publishers' recent writes. 0.5% of these reads would have returned stale data without the publisher's Ticket. Six use cases benefit from global sessions (§ 7.3), totaling 46k writes and 700k reads per second. They often set large write quorums to optimize for read availability and latency.

8.5 Experience and lessons learned

An early lesson was that identifying the appropriate user for a web request was much more difficult than we originally expected. Request endpoints may be invoked before login or after logout; internal applications may track user contexts using bespoke mechanisms; and applications may involve multiple identities, such as when a user manages a business account. Getting high query coverage involved a lot of manual work to discover alternate user contexts and identify endpoints that are not expected to be associated with a user.

Global sessions tend to be used for metadata stored in TAO, such as for product flows modeled as state machines. The addition of global sessions to a code base is often done fairly late in the product development cycle, to fix issues neglected in the initial design. Our ability to strengthen write visibility for such call sites, without data migrations or schema changes, is an important part of making RYW a reasonable default.

The applications that cause the most challenge operationally need RYW consistency the least. These tend to be internal applications that perform batch processing or involve massive fan-out. They often cause write hot spots in the data stores but rarely read what they wrote afterwards.

Closing consistency loopholes with FlightTracker revealed the underlying systems were not actually eventually consistent. We have found low-probability bugs that cause permanent inconsistencies in TAO, graph indexes, and even database replication. These bugs were previously difficult to notice, as they were outnumbered by transient inconsistencies. Ticket-inclusive reads should never return old data, so now that we have FlightTracker even a single occurrence of a stale result is actionable. Bugs leading to permanent inconsistencies included protocol flaws, incorrect handling of error conditions, and relying on data invariants that were not honored by all historical data.

9 Limitations and future work

Our approach still relies on region-sticky user routing. We could avoid this limitation by always using global quorums like in § 7.3, but this would increase latency. We plan to eventually make user RYW sessions global by maintaining a map from user to region in FlightTracker, rehoming sessions when the mapping changes.

The relative efficiency of our solution depends on amortizing the cost of the metadata reads across many TAO queries, and depends on the set of writes being relatively small. Environments with fewer reads have a different set of tradeoffs. This limitation is less applicable to index queries, because those tend to do more work per operation.

FlightTracker does not provide consistency for "unacknowledged successes." As described previously in § 5.3, unacknowledged successes happen when a data write has a client error like a timeout or if the metadata write fails. We have not seen this to be a problem in practice, probably because the issue exists even without FlightTracker.

Some queries are difficult to repair: TAO top-N queries for large edge lists result in unnecessary consistency misses; index queries to a materialized aggregation (such as counts) can be detected as stale by FT-RI, but the stale result cannot be fixed with read repair; and list intersection queries that involve more than two lists are also difficult to repair.

The FlightTracker service compacts Tickets into a timestamp bound, so that we will take a consistency miss if replication exceeds the global compaction bound of 60s (§ 4.2). The global compaction bound is not fully rolled out as of publication time, so tail latency events in the replication pipeline can result in RYW violations. This has not been a big issue in practice because it requires that the first read of a write occurs after the global compaction interval but before replication.

Although some of our motivations are specific to Facebook's workload, our desire to provide user-centric sessions is widely shared [42,55], as is our desire to extend consistency guarantees to global indexes [3,12,30,34]. Cache invalidation is also a perennial challenge for systems at all scales.

Our FlightTracker approach is generalizable: designed for heterogeneous data stores, Tickets can easily be extended to other data stores without much overhead (§ 4.1); the API extensions data stores need to implement (Figure 3) do not require core replication protocol changes and have relatively small overhead (§ 8.3); the client library where a lot of the FlightTracker logic lives can be implemented and rolled out gradually. Our approach is especially beneficial when trying to retrofit indexing systems, because it allows us to separate the reverse metadata index into its own component.

10 Related Work

Stronger consistency atop eventually consistent stores: Many eventually consistent systems offer options to opt for stronger consistency levels. Systems such as Cassandra [37], Riak [6], and RedBlue [39] provide strong consistency either by routing read requests to the leader or by adjusting their commit protocols. To provide bounded staleness, Azure CosmosDB [21] could backpressure writes. In contrast, FlightTracker serves most reads from a single local replica.

Index consistency: Most of these systems, including Amazon's DynamoDB [11, 12] and Google AppEngine Datastore [31], do not extend the stronger consistency levels to global secondary indexes. Twitter's Manhattan [49] extends RYW to global secondary indexes by including them in a cross-shard transactional write, doubling latency [34, 55]. Couchbase [3] supports RYW for reads to its global indexes using a timestamp in the client session. It accomplishes this by deterministically merging updates to all shards, limiting scalability in the number of shards.

Bailis et al. [15] proved that index consistency can be implemented with better availability characteristics than approaches that include indexes in general-purpose transactions.

Implementing RYW sessions: Session RYW [51] is intuitive and implementable with low overhead [14, 27]. Bayou [28] and Pileus [52] provide session guarantees that span multiple servers by managing the session state in their client libraries. Bermbach et al. [19] similarly observed that client-centric consistency should focus on end users; their approach nonetheless assumes that a session is sticky to a single application server. PathStore [42] address the same challenge where clients interact with multiple data store replicas by using a session migration protocol on *every* replica switch. In contrast, FlightTracker manages session state in an intermediate layer between the client and the data servers.

Write-set tracking for stronger consistency: Systems like COPS [40] and SwiftCloud [56] track dependent write sets to provide causal consistency. They also provide client contexts that are similar to FlightTracker sessions. BoltOn [16] layers causal consistency guarantees via a shim over eventually consistent stores. Its design shares similar principles as FlightTracker, aiming to retain the desirable properties of eventually consistent stores. For these systems, the dependency sets need to be stored in the database and cached on the client-side.

TxCache [46] provides transactional (but possibly stale) consistency for application-level caching and uses the terms *staleness miss* and *consistency miss* (both of which are included in our use of the term consistency miss). Its design focuses on single datacenter and treats materialized views as cacheable results from user-specified functions, which is insufficient for our applications.

To reduce the metadata size, systems like Occult [41] and Wukong+S [57] use structural or temporal properties to compress write sets and vector timestamps. FlightTracker uses CRDT inflation to compact Tickets and trims irrelevant writes to reduce network overhead, but mainly avoids metadata size explosion by providing a weaker consistency level.

Tradeoff between cache hit rate and consistency: Zanzibar [44] is built on top of Google's linearizable Spanner [26], but chooses to expose a weaker consistency model to clients to improve its read efficiency and latency. Its zookies play a similar role to FlightTracker Tickets, encapsulating consistency information, but they are used only by Zanzibar itself.

CRDT quorum protocols: The single-round FlightTracker protocol is at its core CRDT [50] using quorum replication [29]. Gryff [22] and CURP [45] similarly leverage commutativity of writes. Because FlightTracker does not need atomicity, a single round suffices.

11 Conclusion

This paper introduces FlightTracker, our approach for providing RYW consistency for Facebook's social graph. FlightTracker operates in a read-optimized ecosystem of asynchronously replicated caches, database replicas, and indexes. It preserves the read efficiency, hot spot tolerance, and loose coupling benefits of eventual consistency, and it has allowed us to circumvent the scaling challenges we encountered when using write-through caching for consistency.

Acknowledgements

We thank the following people for critical contributions to the systems in this paper: Tina Park, Kevin Ventullo, Christopher Small, Andrew Bass, Tushar Pankaj, David Goode, Soham Shah, Lu Pan, Gordon (Zhuo) Huang, Brendan Forsyth, Neil Wheaton, Shilpa Lawande, and Tony Savor.

We thank our shepherd Malte Schwarzkopf and our reviewers for raising the bar on feedback quality. We thank Wyatt Lloyd, Mahesh Balakrishnan, and Rob Lyerly for their many valuable suggestions on the paper.

References

[1] Celery: Distributed Task Queue. http://www.celeryproject.org/.

[2] CockroachDB. https://www.cockroachlabs.com/.

[3] Couchbase Global Secondary Indexes. https://docs.couchbase.com/server/6.5/learn/services-and-indexes/indexes/global-secondary-indexes.html.

[4] LogDevice. https://logdevice.io/.

[5] Resque. http://resque.github.io/.

[6] Riak. https://riak.com/products/riak-kv/.

[7] Three and a Half Degrees of Separation. https://research.fb.com/blog/2016/02/three-and-a-half-degrees-of-separation/.

[8] Thrift. http://thrift.apache.org/.

[9] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing 9*, 1 (1995), 37–49.

[10] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to Adopting Stronger Consistency at Scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (USA, 2015), HOTOS'15, USENIX Association, p. 13.

[11] AMAZON. Design Patterns Using Amazon DynamoDB. https://www.slideshare.net/AmazonWebServices/design-patterns-using-amazon-dynamodb.

[12] AMAZON. Improving Data Access with Secondary Indexes. https://docs.amazonaws.cn/en_us/amazondynamodb/latest/developerguide/SecondaryIndexes.html.

[13] ANTONOPOULOS, P., BUDOVSKI, A., DIACONU, C., HERNANDEZ SAENZ, A., HU, J., KODAVALLA, H., KOSSMANN, D., LINGAM, S., MINHAS, U. F., PRAKASH, N., PUROHIT, V., QU, H., RAVELLA, C. S., REISTETER, K., SHROTRI, S., TANG, D., AND WAKADE, V. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1743–1756.

[14] BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow. 7*, 3 (Nov. 2013), 181–192.

[15] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Coordination Avoidance in Database Systems. *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 185–196.

[16] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, Association for Computing Machinery, p. 761–772.

[17] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 776–787.

[18] BAQUERO, C., ALMEIDA, P. S., CUNHA, A., AND FERREIRA, C. Composition of State-based CRDTs. *HASLab, May* (2015).

[19] BERMBACH, D., KUHLENKAMP, J., DERRE, B., KLEMS, M., AND TAI, S. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *2013 IEEE International Conference on Cloud Engineering (IC2E)* (2013), IEEE, pp. 114–123.

[20] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., AND ET AL. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (USA, 2013), USENIX ATC'13, USENIX Association, p. 49–60.

[21] BROWN, M. Consistency Levels in Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[22] BURKE, M., CHENG, A., AND LLOYD, W. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 591–617.

[23] CHO, S., AVERBUKH, R., ZHANG, Y., CARTER, A., AND JAN, J. A. Partial Update: Efficient Materialized View Maintenance in a Distributed Graph Database. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 1477–1488.

[24] COLBY, L. S., GRIFFIN, T., LIBKIN, L., MUMICK, I. S., AND TRICKEY, H. Algorithms for Deferred View Maintenance. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1996), SIGMOD '96, Association for Computing Machinery, p. 469–480.

[25] COLLET, Y. LZ4 – Extremely Fast Compression. https://github.com/lz4/lz4.

[26] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013).

[27] CROOKS, N., PU, Y., ALVISI, L., AND CLEMENT, A. Seeing is Believing: A Unified Model for Consistency and Isolation via States. *CoRR abs/1609.06670* (2016).

[28] EDWARDS, W. K., MYNATT, E. D., PETERSEN, K., SPREITZER, M. J., TERRY, D. B., AND THEIMER, M. M. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 1997), UIST '97, Association for Computing Machinery, p. 119–128.

[29] GIFFORD, D. K. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1979), SOSP '79, Association for Computing Machinery, p. 150–162.

[30] GJENGSET, J., SCHWARZKOPF, M., BEHRENS, J., ARAÚJO, L. T., EK, M., KOHLER, E., KAASHOEK, M. F., AND MORRIS, R. Noria: Dynamic, Partially-Stateful Data-Flow for High-Performance Web Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (USA, 2018), OSDI'18, USENIX Association, p. 213–231.

[31] GOOGLE. Data consistency in Datastore queries. https://cloud.google.com/appengine/docs/standard/java/datastore/data-consistency#query-data-consistency.

[32] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[33] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX annual technical conference* (2010), vol. 8, p. 9.

[34] KATOORU, K. Native Secondary Indexing in Manhattan. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2018/native-secondary-indexing-in-manhattan.html.

[35] KREPS, J., NARKHEDE, N., RAO, J., ET AL. Kafka: A Distributed Messaging System for Log Processing.

[36] KULKARNI, S. S., DEMIRBAS, M., MADAPPA, D., AVVA, B., AND LEONE, M. Logical Physical Clocks. In *International Conference on Principles of Distributed Systems* (2014), Springer, pp. 17–32.

[37] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[38] LAMPORT, L. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58.

[39] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N., AND RODRIGUES, R. Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (USA, 2012), OSDI'12, USENIX Association, p. 265–278.

[40] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, Association for Computing Machinery, p. 401–416.

[41] MEHDI, S. A., LITTLEY, C., CROOKS, N., ALVISI, L., BRONSON, N., AND LLOYD, W. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (USA, 2017), NSDI'17, USENIX Association, p. 453–468.

[42] MORTAZAVI, S. H., BALASUBRAMANIAN, B., DE LARA, E., AND NARAYANAN, S. P. Toward Session Consistency for the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)* (2018).

[43] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual*

Technical Conference (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[44] PANG, R., CACERES, R., BURROWS, M., CHEN, Z., DAVE, P., GERMER, N., GOLYNSKI, A., GRANEY, K., KANG, N., KISSNER, L., KORN, J. L., PARMAR, A., RICHARDS, C. D., AND WANG, M. Zanzibar: Google's Consistent, Global Authorization System. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)* (Renton, WA, 2019).

[45] PARK, S. J., AND OUSTERHOUT, J. Exploiting Commutativity for Practical Fast Replication. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, p. 47–64.

[46] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional Consistency and Automatic Management in an Application Data Cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (USA, 2010), OSDI'10, USENIX Association, p. 279–292.

[47] SALEM, K., BEYER, K., LINDSAY, B., AND COCHRANE, R. How to Roll a Join: Asynchronous Incremental View Maintenance. *SIGMOD Rec. 29*, 2 (May 2000), 129–140.

[48] SCHAFER, D., AND KUENZEL, L. Subscriptions in GraphQL and Relay. https://graphql.org/blog/subscriptions-in-graphql-and-relay/.

[49] SCHULLER, P. Manhattan: Our Real-Time, Multi-Tenant Distributed Database for Twitter Scale. https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.html.

[50] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free Replicated Data Types. In *Symposium on Self-Stabilizing Systems* (2011), Springer, pp. 386–400.

[51] TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND WELCH, B. B. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems* (Washington, DC, USA, 1994), PDIS '94, IEEE Computer Society Press, p. 140–150.

[52] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 309–324.

[53] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, Association for Computing Machinery, p. 1–12.

[54] VOGELS, W. Eventually Consistent. https://queue.acm.org/detail.cfm?id=1466448.

[55] YARMULA, A. Strong consistency in Manhattan. https://blog.twitter.com/engineering/en_us/a/2016/strong-consistency-in-manhattan.html.

[56] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference* (New York, NY, USA, 2015), Middleware '15, Association for Computing Machinery, p. 75–87.

[57] ZHANG, Y., CHEN, R., AND CHEN, H. Sub-Millisecond Stateful Stream Querying over Fast-Evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 614–630.

[58] ZHUGE, Y., GARCÍA-MOLINA, H., HAMMER, J., AND WIDOM, J. View Maintenance in a Warehousing Environment. *SIGMOD Rec. 24*, 2 (May 1995), 316–327.

KVell+: Snapshot Isolation without Snapshots

Baptiste Lepers
University of Sydney

Oana Balmau
University of Sydney

Karan Gupta
Nutanix

Willy Zwaenepoel
University of Sydney

Abstract

Snapshot Isolation (SI) enables online analytical processing (OLAP) queries to observe a snapshot of the data at the time the query is issued, despite concurrent updates by online transactional processing (OLTP) transactions. The conventional implementation of SI creates a new version of a data item when it is updated, rather than overwriting the old version. Versions are garbage collected when they can no longer be read by any OLAP query. Frequent updates during long-running OLAP queries therefore create significant space amplification, and garbage collection can give rise to latency spikes for OLTP transactions. These problems are exacerbated on modern low-latency drives that can persist millions of updates per second.

We observe that analytic queries often consist in large part of commutative processing of data items resulting from range scans in which each item in the range is read exactly once. We introduce Online Commutative Processing (OLCP), a new model for processing analytical queries, that takes advantage of this observation. Under OLCP, analytical queries observe the same snapshot of the data as they would under conventional SI, but space amplification and garbage collection costs are largely and oftentimes nearly entirely avoided. When an item in such a range is updated, the old version of the item is propagated to the OLCP queries that might need it instead of being kept in the store.

We demonstrate OLCP's expressiveness by showing how to formulate, among others, the TPC-H benchmark queries in OLCP. We implement OLCP in KVell+, an extension of KVell, a key-value store for NVMe SSDs. Using YCSB-T, TPC-CH and production workloads from Nutanix, we run a wide range of analytics queries concurrently with write-intensive transactions. We show that OLCP incurs little or no space amplification or garbage collection overhead. As a surprising by-product we also show that OLCP speeds up analytical queries compared to SI.

1 Introduction

The desire to run frequent analytics on fresh data has led to the recent development of databases that allow concurrent processing of online transaction processing (OLTP) and online analytical processing (OLAP) [34]. To isolate OLAP queries from OLTP updates, databases typically rely on Snapshot Isolation (SI) [51, 66, 69]. SI provides OLAP queries with a snapshot of the database at the time the query is issued, independent of later updates made by OLTP transactions. Conventionally, SI is implemented by multi-versioning [6]: an update generates a new version of a data item, and previous versions are kept for as long as they belong to an active OLAP query's snapshot. Versions that no longer belong to such a snapshot are garbage collected. Long queries may thus cause the store to grow—a phenomenon known as *space amplification*, and garbage collection may provoke latency spikes.

Minimizing disk usage is important in production systems. Facebook found that "storage space is the bottleneck" [23], and Alibaba Group runs garbage collection with "the highest priority to prevent waste of storage space" [32]. Space amplification is particularly problematic when the dataset is stored on modern storage devices. NVMe SSDs can persist millions of items per second. Furthermore, because random and sequential access bandwidth are nearly identical, scanning data is no longer faster than performing random access updates. An analytical query running concurrently with write-intensive transactions may therefore cause the size of the store to increase manyfold.

Space amplification is a well-known problem for in-memory SI data stores. Various solutions have been developed, but they perform poorly on disk-based systems. Executing transactions sequentially avoids the need for locking and versioning [28], but is impractical when I/O latencies have to be overlapped with CPU use. Creating snapshots using operating system fork and copy-on-write techniques [39] incurs very high file system overheads when applied to disk-based systems. Closest to our work, Steam [8] trims versions that do not belong to any *active* snapshot, providing efficiencies

for some workloads. We propose a more radical re-design, suitable also for disk storage, that seeks to altogether avoid keeping old versions in the store.

Our approach is based on the following two observations. First, most OLAP queries scan data, but are oblivious to the order in which they read items, because the operations performed on items are commutative. Second, OLAP queries read scanned items at most once. For instance, queries that compute sales statistics (e.g., the most popular item in a region) can perform their operation by scanning items once in any order. Based on these observations, we define a new class of processing: OnLine Commutative Processing (OLCP). OLCP queries declare so called *scan ranges*. When an item in a scan range is updated by a concurrent transaction, its old value is processed by OLCP queries and then discarded, instead of being kept in the store.

Scan ranges have numerous advantages. First, because no versions are kept for items in scan ranges, space amplification is limited, and GC overhead is reduced. Second, because OLCP queries process items in scan ranges as they are modified, they read more data from memory and less from disk, and thus have higher throughput than their OLAP counterparts. The trade-off is that an OLCP query can read items belonging to its scan ranges only once. In addition, scanned items are not guaranteed to be read in order.

In addition, OLCP queries can declare *point ranges*, ranges of items on which they want to perform point queries. Items in point ranges are versioned, as in the conventional SI implementation. The combination of scan ranges and point ranges allows OLAP queries to be expressed efficiently in OLCP. Moreover, a very large subset can be expressed in a manner so that they derive great benefit from OLCP, including reduced space amplification, no GC-induced latency spikes, and higher throughput.

We implement OLCP queries in Kvell+, an extension of KVell [45]. In KVell+, scan and point ranges are declared through an interface inspired by the MapReduce paradigm [19]. OLCP queries declare a `map` function that is called *exactly once* on all items that belong to scan ranges. The items that `map` reads correspond to the items that the query would have read under conventional SI (i.e., belonging to the snapshot at the start of the query). The `map` function can also perform point queries on items in point ranges. In the absence of updates by OLTP transactions, `map` is called on items in scan ranges in lexicographic order, but when an item is updated, we *propagate* its old value to OLCP queries. The old value is processed by the OLCP queries (potentially breaking the lexicographic order of the scans) and then deleted from the store. Space freed by the deletion can be reused to store new items.

OLCP can easily be integrated in existing applications, either manually, using an SQL-to-MapReduce tool [44,71], or automatically at the SQL query-plan level. OLAP and OLCP queries can run simultaneously on the same data. A developer may therefore choose to port existing OLAP queries that create substantial space amplification to OLCP, while leaving less problematic OLAP queries to run under SI.

We make the following contributions:

- The OLCP query model.
- A detailed explanation and examples showing how to port OLAP queries (e.g., MapReduce analytics, TPC-H queries) to OLCP.
- The implementation of OLCP in KVell+.
- A comparison of OLCP to SI and Steam [8] in terms of space amplification, tail latency and throughput.

Roadmap. Section 2 presents the key OLCP principles. Section 3 explains in detail how data analytics workloads can greatly benefit from OLCP. Section 4 discusses the implementation. Section 5 shows our experimental evaluation results. Section 6 presents the related work, and Section 7 concludes.

2 OLCP Overview

In this section, we explain OLCP's design principles, advantages, and limitations. We explain how to perform scans concurrently with propagation events.

2.1 OLCP in a nutshell

The main goal of OLCP is to reduce the time that old item versions spend in the store. OLCP allows the store to *reduce the lifespan of old versions down to the duration of OLTP commits*. In a conventional SI implementation, OLAP queries force the store to keep old versions for the entire duration of the queries. In contrast, OLCP queries process old versions as they are generated. Once the old version of an item has been processed, it is deleted from the store and its space can be reused to store new items.

OLCP advantages: OLCP provides the same guarantees as SI, with virtually no space amplification. Furthermore, because OLCP queries process items as they are being updated, OLCP queries avoid reads to disk, improving the throughput of analytical queries.

OLCP requirements: To completely avoid space amplification under OLCP, queries need to support scanning out-of-order and to access each item in the scan ranges only once. These requirements can be relaxed at the expense of increased space amplification, but OLCP's space amplification is always lower than that of conventional SI implementations. OLCP queries are widely applicable and constitute an efficient replacement of OLAP queries, as we demonstrate in Section 3.

2.2 OLCP interface

MapReduce interface: The interface of OLCP is inspired by the event-driven MapReduce paradigm [19]. An OLCP query is created and executed by a single function call:

```
t = olcp_query(map, payload, [scan_range_1,
  scan_range_2, ...], [point_range_1, ...])
```

The `olcp_query` call takes the following parameters:

- **A `map` function callback.** OLCP guarantees that the `map` callback is called exactly once on all items within the scan ranges. The exactly-once guarantee is essential to limit overheads. Without it, OLCP would have to maintain a list of items they have already seen, which could have prohibitive CPU and memory overhead for large scans. The item versions provided to the `map` callback correspond to those that the query would have scanned under SI (i.e., those belonging to the snapshot at the time the query is launched).
- **Payload for the `map` callback.** An arbitrary pointer to application specific data. Usually used to retrieve or store intermediary computation results.
- **Scan ranges.** This range can be the entire store. If ranges overlap, the `map` function is called only once per item belonging to the ranges. Items belonging to the ranges are not versioned and induce no space amplification. In return, items belonging to the ranges are not guaranteed to be scanned in order (old versions might be scanned before their turn to avoid keeping them in snapshots).
- **Point ranges.** OLCP queries may also declare ranges of items that they might access using point queries. Items within those ranges are versioned until the query is committed and may induce space amplification. Items in the point ranges can be accessed multiple times, and scans on these ranges are guaranteed to happen in lexicographic order. Many analytical processing queries can be expressed without using point queries, as we show in Section 3.

Items outside of the scan and point ranges are neither versioned nor propagated. The `olcp_query` function blocks until the scans are complete. After calling `olcp_query`, a developer might choose to do further processing on the payload. In the remainder of this paper, we do this processing in a `reduce` function.

2.3 Scans, propagation and space reclamation

Algorithm 1 presents pseudo code for scanning, updating and propagating updates in a store that supports OLCP queries. For simplicity, we present a sequential implementation that does not support point ranges. A full implementation would have to handle possible races between scans and propagations and delay the deletion of items belonging to point ranges. We also assume the use of timestamps to define snapshots, as is common in SI implementations.

Algorithm 1 Pseudo-code of a sequential implementation of updates, propagations, and scans.

```
1  /*OLCP commit: create a new version and add the
2  old version in GC queue */
3  timestamp t_commit = now();
4  active_commit_timestamps.add(t_commit);
5  foreach(item i in updated_items) {
6      kv.write(i, t_commit);
7      gc.add(get_oldest(i), t_commit);
8  }
9  active_commit_timestamps.delete(t_commit);

11 /* GC */
12 timestamp t_min=min(active_commit_timestamps);
13 foreach(item i in gc) {
14     // Only delete items from
15     // fully committed transactions
16     if(i.t_commit >= t_min)
17         break;
18     foreach(olcp o in running_olcp) {
19       if(o.in_snapshot(i)
20         && i.key > o.last_scanned)
21          o.propagation_queue.add(i);
22     }
23     delete(i); // remove from the store
24 }

26 /* OLCP query thread */
27 item last_scanned = get_first(scan_range);
28 do {
29   if(last_scanned != EOF) {
30     map(last_scanned, payload);
31     get_next(&last_scanned);
32   }
33
34   while(item i = propagation_queue.pop())
35     if(i.key > last_scanned)
36         map(i, payload);
37 } while(last_scanned != EOF);
```

Scans: OLCP queries request items from the store in lexicographic order using the `get_next` function (line 31 of Algorithm 1). When there are no concurrent OLTP transactions, the scan happens as it would in a conventional SI implementation: items are read in lexicographic order, and the map function is called on each of them. In OLCP, however, this order can be "interrupted" by propagations resulting from updates by OLTP transactions to items in the scan ranges. When receiving a propagated item, the OLCP query checks that it has not yet scanned the item and, if so, calls map on it (lines 35-36). Afterwards, the OLCP query resumes the scan from the last scanned item using the `get_next` function.

Propagation and space reclamation: The key to avoiding space amplification with OLCP queries is to delete old data as soon as possible. However, an old version of an item cannot be deleted as soon as a new version is created. When an OLTP transaction updates multiple items, old items can be deleted

only after *all* new items are persisted (to allow recovery in case of a mid-commit crash). Hence, the deletions must happen *after* a transaction has committed. Consequently, the store must maintain multiple versions of committed items for the duration of an OLTP commit.

Committing an OLTP transaction then consists of updating the modified items in the store and enqueueing the oldest version of those items on the GC queue (lines 6-7 of Algorithm 1). After the commit is completed, the GC propagates and deletes items. An item is propagated to an OLCP query only if it belongs to the query's snapshot and if it has not yet been scanned. We rely on the lexicographic order of the scan to efficiently ensure this latter property (line 20). In our pseudo code, we choose to enqueue propagated elements in a per-OLCP query queue (line 21), but an implementation might choose a different communication mechanism between the store and running OLCP queries.

Space reclamation efficiency: In practice, the number of versioned items in OLCP is small. When an OLCP query does not use point ranges, a rough estimate of the number of versioned items in OLCP is the number of updates per transaction times the number of concurrent commits. In a conventional implementation of SI, this number is much higher, since the system needs to keep old versions of all items updated during the lifetime of OLAP queries.

Old versions are also only kept for a much shorter time in OLCP. Figure 1 summarizes the lifespan of objects, executing an OLTP transaction concurrently with an (a) OLAP or (b) OLCP query. With OLAP queries, the store has to keep all versions of items for the duration of long queries (minutes), while OLCP allows the store to remove old versions after at most a few commits (microseconds).

OLCP queries can run alongside OLAP queries. In that case the deletion and propagation of items is postponed to ensure the correct execution of OLAP queries: items are deleted and propagated when they no longer belong to an OLAP snapshot. OLAP queries may thus reduce the effectiveness of using OLCP.

2.4 Informal correctness argument

Correctness requires that, despite concurrent OLTP transactions, OLCP queries read the same items from their scan ranges as they would have read under a conventional implementation of SI, and that these items are processed *exactly once*. The correctness relies on the following observations.

An item is propagated at most once, and the propagated item belongs to the query's snapshot. If an item is not updated, then no propagation occurs. If an item is updated once, its old version is propagated only if it belongs to the snapshot (line 19 in Algorithm 1). If an item is updated multiple times, all old versions are put in the GC queue, but only one of them belongs to the snapshot and is propagated.

An item is processed exactly once. If an item is not propagated, it is read as part of the scan. The scan does not "skip" items: after scanning an item, a query always requests the next item from the store regardless of concurrent propagations. Thus, a query always scans its entire *scan_ranges*. Only items that have not yet been scanned are propagated (lines 20 and 35 in Algorithm 1).

From the previous observations, we conclude that an OLCP query processes all the items belonging to its scan ranges exactly once, and that the processed items belong to its snapshot. As a result, a developer need not consider the distinction between scanned and propagated items.

2.5 Example

Figure 2 illustrates with an example some of the complex interleavings between OLTP and OLCP. An OLCP query T scans a range of 5 items. T has snapshot timestamp 0. The initial versions of all five items have timestamp 0, and therefore belong to T's snapshot. Despite various updates by OLTP transactions, T correctly calls `map` exactly once on all five initial item versions.

Of particular interest in this execution is item d that is updated twice, at t_2 and t_4, but only d_0 is propagated. Despite being interrupted by the propagation of d_0, T correctly

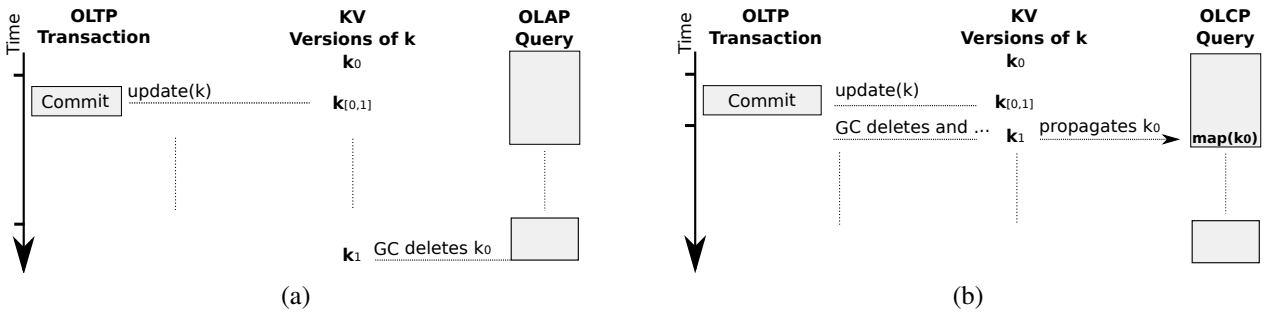


Figure 1: Lifespan of items under OLAP (a) and OLCP (b). OLAP forces an old item version to be kept for the entire duration of the OLAP query, while OLCP needs to keep it only for the duration of the commit of the OLTP transaction that produces the new version.

resumes its scan from b_0 at time $t3$. Finally, at t_5, a is not propagated because the query already scanned b, and at t_6, c is not propagated because it has just been scanned.

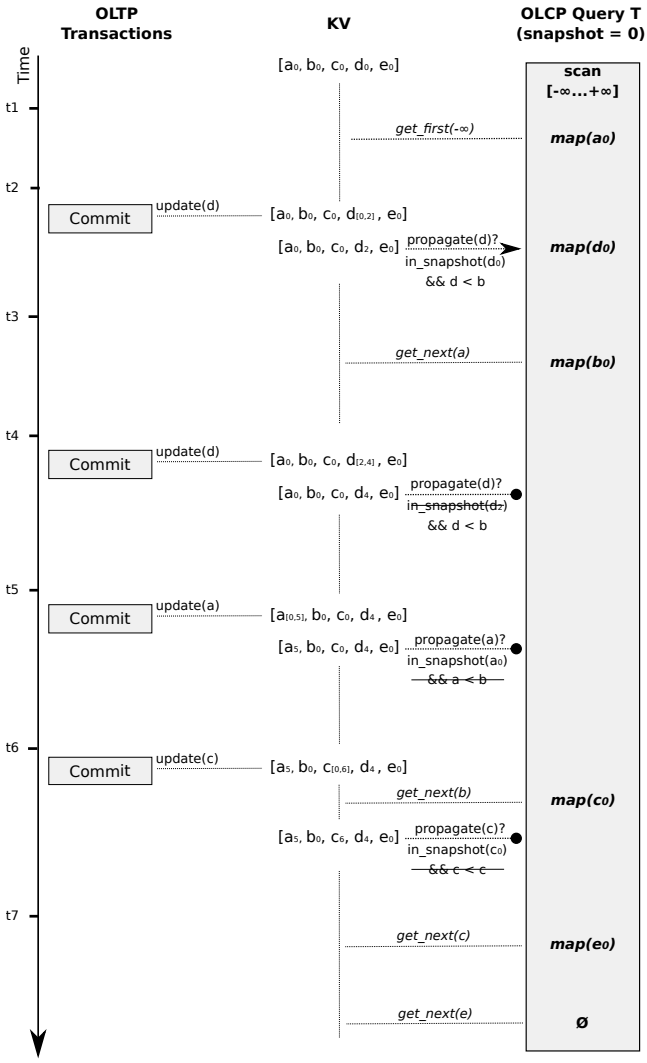


Figure 2: Possible interleavings between a scan and various propagations. Arrows indicate that an item is propagated to the OLCP query, rounded segments indicate that an item is not. At the end, the `map` function has been called exactly once on all items initially contained in the scan range.

3 Using OLCP in practice

Below, we explain how OLCP can be widely used in practice to eliminate space amplification. Analytical processing is typically done in the following three ways:

1. Multidimensional OLAP (MOLAP) analytics,
2. Relational OLAP (ROLAP) analytics, and
3. MapReduce-style analytics.

3.1 MOLAP analytics

MOLAP databases provide a traditional platform for data analytics which is widely used in Business Intelligence applications (e.g., IBM Cognos [14], Oracle Essbase [56], and iccube [35]). The data is first extracted from a relational database, transformed into a specialized multidimensional cube format, and then transferred into the MOLAP database. OLCP can be used during the extraction phase to get a snapshot of the database with little space overhead. Using a conventional implementation of SI, updates performed during the extraction create space amplification, and the relational database may stall once the extraction is complete because of GC. To avoid these issues, database administrators usually run the extraction during the night when the load is low. OLCP allows extractions to occur at any time without space overhead or database stalls.

3.2 ROLAP analytics

ROLAP tools query the main relational database directly through a language like SQL. In this paper we use SQL syntax for simplicity, but other languages with similar constructs can be used as well. Analytical queries consist of a combination of three types of building blocks.

1. Decomposable aggregate functions (e.g., `SUM`, `COUNT`).
2. Aggregate functions (e.g., `GROUP BY`, `CUBE`, `ROLLUP`).
3. Joins.

We show that for these three types of operations OLCP reduces space amplification. *Most ROLAP operations have no space overhead under OLCP.*

Decomposable aggregate functions: Decomposable aggregate functions are the least complex of the three query building blocks. They consist of commutative operations that only require one pass over the data (e.g., `SUM`, `MIN`, `MAX`, `AVG`).

Nutanix uses decomposable aggregate functions to compute simple statistics on a store that keeps track of disk blocks allocated to virtual machines in a datacenter. For instance, Algorithm 2 counts the number of disk blocks that have not been accessed for the last two hours. The query is used to estimate the percentage of allocated storage that is infrequently accessed. Algorithm 3 presents the equivalent using OLCP. For simplicity, we present a sequential version of the algorithm. In practice the `map` function can be called concurrently by multiple threads, and we use per thread payloads that are merged at the end of the scan.

The query is executed with low priority in order to avoid interfering with other workloads. This query used to be executed under read committed to avoid the space amplification overhead of SI (under read-committed, old data is removed from the store and the scan reads the most recent version of committed items). Unfortunately, under read committed,

this query overestimates the number of accessed blocks because, when a block is accessed after the start of the query, it is impossible to know if the block was idle in the past two hours prior to the query, since this information is lost after the update. Executing the query with OLCP, and thus with SI guarantees, produces a precise estimate of disk usage.

Algorithm 2 RocksDB pseudocode for counting the number of disk blocks in a datacenter that have not been used in the last two hours.

```
size_t count = 0, target_time = now() - 2;
Iterator *it = ...;
for(it->SeekToFirst();it->Valid();it->Next()){
      block_t *t = it->value;
      if(t->last_access < target_time)
       count++;
}
```

Algorithm 3 OLCP pseudocode for counting the number of disk blocks in a datacenter that have not been used in the last two hours.

```
map(item *i, payload *p) {
    if(i->last_access < p->target_time)
        p->count++;
}
payload p={.target_time = now()-2, .count=0};
t = olcp_query(map, &p, [...], NULL);
commit(t);
```

Aggregate functions: Like decomposable aggregate functions, these queries do not require items to be accessed in order, and the data is accessed only once. The difference is that these queries group items into categories and compute statistics for each group (e.g., using the `GROUP BY` clause and its extensions like `CUBE` and `ROLLUP`).

We illustrate how OLCP reduces space overhead with the first query from the TPC-H suite [74] (Algorithm 4). Algorithm 5 presents its equivalent using OLCP (for simplicity we use the name of the tables to represent the ranges of keys). The query provides a summary pricing report for all items shipped before a given date, aggregated by a flag and a status. This query is more complex than the previous example because it requires grouping analyzed items in buckets and returning them in order. Since the number of flags and statuses is small, the number of buckets is small, and the summaries can be computed in memory. If the number of summaries to be computed was large, the `map` function could use point queries to load and store temporary summary results from disk. After the scan completes, the summaries are sorted in a `reduce` function.

Joins: ROLAP joins are typically *hash joins* or *nested loop joins* [30]. An analysis of the query plans of Microsoft SQL [12] for the TPC-H queries shows that approximately 70% of the joins are hash joins, and the remaining 30% are nested loop joins.

Algorithm 4 First query of TPC-H.

```
select l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty, [...]
from lineitem
where l_shipdate <= '1998-09-04'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

Algorithm 5 First query of TPC-H using OLCP.

```
map(item *i, payload *p) {
  if(i->l_shipdate < "1998-09-04")
     return;
  string k=i->l_returnflag+"|"+i->l_linestatus;
  p->sum_qty[k] += i->l_quantity;
}

reduce(payload *p) {
   sort(p->sum_qty); // sort p by key
   return p->sum_qty;
}

payload p = { ... };
t = olcp_query(map, &p, [lineitems], NULL);
commit(t);
reduce(&p);
```

Hash joins are usually performed in two steps. First, the join scans the first table, and builds a hash table (build phase). Then, the join scans the second table and probes the hash table for matches (probing phase). Building the hash table only uses one-time commutative reads. By putting the first table in the scan ranges, OLCP avoids any space amplification during the build phase. The probing phase then occurs in the *reduce* function, with the second table in the point ranges. In general, hash joins can easily be ported to OLCP by placing the more frequently updated table in the scan ranges and the less frequently accessed table in the point ranges.

Algorithm 6 presents the fourth query of TPC-H. The query counts the number of orders ordered in a given quarter of a given year in which at least one lineitem (item of an order) is received by the customer later than its committed date. In Microsoft SQL, the build phase is performed on the "orders" table and the probing phase on the "lineitem" table. Algorithm 7 presents a port of this query plan to OLCP. The "orders" table is placed in the scan ranges, and the "lineitem" table in the point ranges. The hash table is built in the `map` function, and the scan of lineitem and the probing is done in the `reduce` function. While the query is running, updates on the "orders" table, or any table that is not accessed by the query, do not induce any space amplification. Items in the "lineitem" table are versioned. If "lineitem" were known to be frequently updated, the query plan could easily be modified to build the hash table using "lineitems" and scanning the "orders" table next.

Algorithm 6 Query 4 of TPC-H in SQL.

```
select  o_orderpriority, count(*) as order_cnt
from orders
where
  o_orderdate >= date '1995-01-01'
  and o_orderdate < date '1995-04-01'
  and exists (
    select *
    from lineitem
    where l_orderkey = o_orderkey
      and l_commitdate < l_receiptdate
  )
group by o_orderpriority
order by o_orderpriority;
```

Algorithm 7 Pseudo code of Query 4 of TPC-H using OLCP.

```
map(item *i, payload *p) {
  if(i->o_orderdate >= '1995-01-01'
    && i->o_orderdate < '1995-04-01')
    p->hash[i->o_orderkey] = ...;
}
reduce(payload *p) {
  // Scan versioned lineitems
  hash_t res = {};
  foreach(lineitem_t l in lineitems) {
    if(p->hash[l->l_orderkey]
        && l->l_commitdate < l->l_receiptdate)
      res[l->o_orderdate].order_cnt++;
  }
  return sort(res);
}
payload p = { ... };
t=olcp_query(map,&p,[order],[lineitem]);
reduce(&p);
commit(t);
```

Nested loop joins iterate over two tables in order. Because of the order constraint, nested loops do not naturally fit the OLCP model. However, it is often possible to adapt nested loops to OLCP with minor changes to the query plan. Query 17 of the TPC-H benchmark (Algorithm 8) is an example of a complex join query that is executed using nested loops in the Microsoft SQL query plans for TPC-H. This query gets items of a given brand that sold five times less than the same item from other brands. It then computes the total revenue loss that would have occurred if these items had not been sold. The query is divided in two sections: *tinner* computes the average number of sales per "partkey" item, regardless of the brand, and *touter* gets the sales information for a given brand.

The number of "partkey" items is small (10K) compared to the number of order items (90M), and orders are aggregated by "partkey". Algorithm 9 presents pseudo code of a possible implementation. Lineitem (list of ordered items) is scanned, and the `map` function simultaneously computes information for the *tinner* and *touter* queries. The `map` function performs one point query to the "partkey" table to get the brand of the scanned item. A `reduce` function then aggregates per-partkey information and outputs the total price of the items that match the criteria. The memory required to execute this query is low (hashtable with 10K entries). The "partkey" table is the only table that is accessed using a point query. Since "partkey" is read-mostly, this query has negligible space amplification when executed with OLCP.

Algorithm 8 Query 17 of TPC-H.

```
select sum(l_extprice) / 7.0 as avg_yearly
from
  (
    select l_partkey, l_quantity, l_extprice
    from lineitem, part
    where p_partkey = l_partkey
      and p_brand='Brand#34'
      and p_container='MED␣PACK'
  ) touter,
  (
    select l_partkey as lp,
       0.2*avg(l_quantity) as lq
    from lineitem
    group by l_partkey
  ) tinner
where touter.l_partkey = tinner.lp
 and touter.l_quantity < tinner.lq;
```

Algorithm 9 Pseudo code of Query 17 using OLCP.

```
map(item *i, payload *p) {
  string k = i->l_partkey;

  // Tinner
  p->tinner[k].l_quantity_sum += i->l_quantity;
  p->tinner[k].l_quantity_count++;

  // Touter
  part_t *part = kv_get("part"+k); // Seek
  if(part->p_brand == "Brand#34"
    && part->p_container = "MED␣PACK") {
    p->touter[k] += {
        l_quantity = i->l_quantity,
        l_extprice = i->l_extprice
    };
  }
}
reduce(payload *p) {
 double l_extprice_sum = 0;
 foreach(string k in p->touter) {
   double lq = 0.2*p->tinner[k].l_quantity_sum/
     p->tinner[k].l_quantity_count;
   foreach(int i in p->touter[k]) {
     if(p->touter[k][i].l_quantity < lq)
       l_extprice_sum+=p->touter[k].l_extprice;
   }
 }
 return l_extprice_sum / 7.0;
}
payload p = { ... };
t=olcp_query(map,&p,[lineitems],[parts]);
commit(t);
reduce(&p);
```

3.3 MapReduce analytics

MapReduce provides a highly parallelizable and scalable framework. This approach is popular for computing simple analytics on vast amounts of data, employed for instance to obtain cluster management statistics [9], to compute popular search-word and query trends [20], and to analyze time-series workloads in IoT, recommender systems and finance [1]. Essentially, the mappers are doing a background scan on the store (e.g., Cassandra, RocksDB), and push the items of interest into a MapReduce system (e.g., Hadoop, CouchDB, Phoenix [52]). These scans take on the order of a few hours and happen concurrently with the foreground workloads, which can be write-heavy [3]. Using the conventional implementation of SI causes prohibitive space amplification because incoming updates need to be tracked over a long time span. To avoid the space explosion, these statistics are usually collected in read-committed mode and thus have lower accuracy. In contrast, OLCP supports consistent one-pass scans, with no space overhead.

4 Implementation

In this section, we describe our implementation of SI and OLCP. The source code of our implementation is available at *https://github.com/BLepers/KVell*. Our implementation adds approximately 4,000 lines of code on top of KVell.

4.1 KVell

As noted in previous work [45], when running on modern fast drives, existing KVs that support SI, such as WiredTiger and RocksDB, run into a CPU bottleneck and are unable to write data at disk speed. As a result they are not suitable for studying space amplification on such drives. We therefore extend KVell [45], a recent KV designed for NVMe SSDs.

KVell has two main components: an ordered index residing in RAM, and an unsorted data structure on disk similar to a slab memory allocator, which groups items with similar sizes in the same file. Reads are either served from a cache (0 I/O), or from disk (1 I/O). Updates fetch a 4KB block from disk, modify it in memory, and then write the dirty block back to disk (1 or 2 I/Os, depending on whether the block was cached or not). The index and the disk data structure are partitioned among multiple worker threads, with each worker handling a range of the key space.

Ideally, analytical queries should not slow down OLTP transactions. Even on modern drives, a fine balance has to be maintained between sending too few simultaneous requests (resulting in sub-optimal bandwidth) and sending too many (resulting in high latency). In its original implementation, KVell scans ranges by reading all items of the range in parallel. We change the implementation of scans to ensure that scans do not overwhelm the disk with with requests. Scans request batches of items from the store, with the size of a batch adjusted depending on the current disk utilization. In practice, we aim at having between 32-64 pending disk I/O requests at all time. When reading the next batch, we adjust the batch size to keep the number of disk I/Os within this bound.

In its original implementation, KVell did not support transactions. We first describe our conventional implementation of SI in KVell+ and the extension to reduce space amplification proposed in Steam [8]. We then describe our implementation of OLCP in KVell+.

4.2 Conventional SI

Our implementation of SI is inspired by those of RocksDB and WiredTiger, two KVs that are widely used in industry.

Timestamps: We add a global logical timestamp in KVell. The global timestamp is incremented every time it is read. When a transaction *commits*, it is given a commit timestamp t_{commit} equal to the current global timestamp. When a transaction *starts*, it is given a snapshot timestamp, $t_{snapshot}$. The snapshot timestamp is chosen so that a transaction can only read data that has already been committed, using the following formula: $t_{snapshot} = min_{active}(t_{commit}) - 1$. If no transaction is committing, the $t_{snapshot}$ is set to the current global timestamp.

In the original version of KVell, persisted items are already timestamped; we use these timestamps in the read and writing path: a transaction can only read or write an item with a timestamp less than or equal to its $t_{snapshot}$.

Writing data: To perform a *write* on a key, a transaction locks the index entry for that key in the main memory index. If the key is not present in the store, a new locked index entry is created. To prevent write-write conflicts, a transaction that fails to lock an item aborts. It also removes all previously acquired locks and any newly created index entries. Before commit, only the in-memory index is updated. The new item versions are kept in a private in-memory buffer (similarly to RocksDB).

Reading data: When *reading* an item, the worker first checks if the item is in its private buffer. If not, the item is read from the main store. If the memory index contains multiple versions of an item, the transaction reads the most recent version that belongs to its snapshot.

Committing updated data: To commit, a transaction persists an tuple "(t_{commit}, N)", where N is the number of updated items. This tuple is used in case of a crash to avoid recovering items from partially committed transactions. The transaction then writes the new items to disk, timestamped with t_{commit}. Once all new items have been persisted, the transaction deletes the "(t_{commit}, N)" tuple. During a commit, the transaction updates the index non-atomically: entries for the new versions are added to the index, and index entries are unlocked as they are updated. This process is safe because no other transac-

tion can read or write any of the updates before the commit ends (by the definition of $t_{snapshot}$), so transactions cannot access partially committed data. Hence, transactions appear "atomically" in the system.

KVell did not use a commit log in its original implementation, and we do not add one to support transactions. This design choice is essential for performance on modern drives. Historically, commit logs were cheap to maintain compared to the cost of updating the store – a fast sequential append vs. a slow random update to a complex data structure. NVMe drives can perform random I/Os as fast as sequential I/Os. In KVell, persisting an item is performed in as low as 1 I/O. Adding a commit log would essentially double the number of I/Os required to perform an update and halve the speed of the store. We acknowledge the usefulness of logs (e.g., for accountability, audit, etc.), and developers might choose to log store accesses via a fast logging system. Our implementation has the advantage of placing logs outside of the critical path.

Garbage collection: After commit, the location of the old versions of updated items are placed in a per-worker cleaning list. Workers periodically check the smallest active $t_{snapshot}$. When this value changes, they scan their cleaning list and delete obsolete items. Workers stop cleaning as soon as they find an item with a timestamp higher than or equal to $min(t_{snapshot})$ (similarly to WiredTiger).

4.3 Steam

Steam [8] uses a more aggressive form of garbage collection that aims to reduce the number of old versions. When an item is updated, Steam scans that item's versions, and deletes the ones that do not belong to any active transaction. Steam was originally implemented in an in-memory database and does not handle recovery in case of a crash. In our implementation, we delay the deletion of old versions to after the commit to avoid deleting versions that might be needed during recovery. Otherwise, our implementation is similar to the original one.

4.4 OLCP in KVell+

OLCP further modifies garbage collection and implements propagation, We also describe a key optimization to avoid extra I/Os as a result of propagation.

Garbage collection: The main difference between OLCP and OLAP is the time during which old versions need to be kept in the store. Workers have two cleaning lists: one for items belonging to the scan ranges, and one for items belonging to the point ranges. GC for point ranges happens as it would under SI. GC for scan ranges happens as described in Algorithm 1.

Propagations: Key to the proper functioning of OLCP is implementing an efficient propagation mechanism. KVell uses an asynchronous interface: threads send requests to the datastore, and the datastore enqueues answers in a per-transaction queue. We build on this mechanism for propagations. Propagating an item I to a OLCP query T consists of enqueuing I in T's queue (as if T had requested to read the item). At the data store level, data is sharded between single threaded workers, so propagations do not introduce any data races. For instance, if an item is propagated "while" being requested by a scan, the scan request and the propagation request are serialized at the worker level and only one of the requests causes the item to be enqueued in the queue. If multiple OLCP queries are running, an item may be enqueued in multiple queues.

Concurrency: In KVell, items are sharded between multiple workers. To speed up queries, we start the scan on all workers. All workers progress in their scan concurrently and may propagate updates concurrently. No synchronization is required between workers because workers work on distinct items. In practice, the scan happens as if multiple single threaded scans were launched on disjoint sets of items.

Avoid reading old versions from disk: In Section 2, old item versions are propagated immediately after committing the new versions. This approach is sub-optimal: updates are not performed in place, and therefore propagating old versions at this time requires reading them from disk, adding an extra read to an update. To eliminate this extra read, we delay propagations and deletions. Instead of propagating and deleting the old versions in the GC (lines 18- 23 in Algorithm 1), we keep the entries for them in the index, and we put their location in a list of reusable spots. When such a spot is later reused, the disk block containing that spot is read, and we take advantage of that to propagate the old version without an extra disk read.

This optimization raises the possibility that versions of the same item might not be overwritten in the order they are created. For instance, if versions of the same data item are of different size, they are allocated in different slabs, and a more recent version may be overwritten before an older version.

Figure 3 presents a case where an item has three versions (k_0, k_1 and k_2). k_2 is the current version. k_0 and k_1 are old versions ($t0 < t1 < t2$). k_0 and k_1 are in the free list of reusable spots and have not yet been overwritten. At the beginning of the execution, the in-memory index still contains all three versions. Indeed, k_0 and k_1 have not been overwritten, and so have not yet been propagated. In Figure 3 an OLCP query T executes with a snapshot timestamp equal to $t1$. Assume that the slot containing k_1 is reused before the one containing k_0, and assume furthermore that k has not been scanned. k_1 is propagated and deleted, since it has not been scanned and it is part of T's snapshot.

However, k_1's index entry must not be immediately removed. In the absence of any record of k_1 in the index, if k_0's slot is overwritten before the scan reaches k, as depicted in Figure 3, it would be propagated to T. Similarly, if T's scan reaches k before k_0 is overwritten, it would be read by the

scan. In both cases, T would erroneously read k_0, a version that does not belong to its snapshot. To avoid these situations, we keep k_1 in the index, but flag it as deleted. It then becomes clear that k_0 does not belong to T's snapshot, and it is neither propagated nor read by the scan. Once k_0 has been overwritten, it is removed from the index because it is the oldest version. k_1 is then removed from the index as well because it is now the oldest version and flagged as deleted.

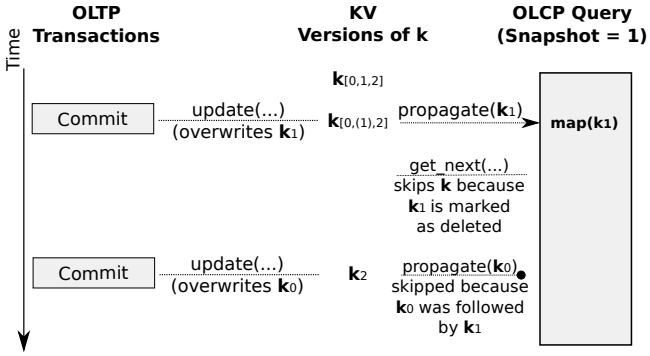


Figure 3: Under optimized OLCP old versions might not be overwritten in version order. In that case, the query must skip k_0, even though it would appear to belong to its snapshot ($t0 < t1$).

4.5 OLCP in other stores

In the previous section, we focused on the implementation of OLCP in KVell, but the OLCP paradigm is general and applicable to other datastores and other storage devices. For instance, OLCP can be implemented in RocksDB and WiredTiger and on slower SSDs. In RocksDB, old items can be propagated during compactions: when merging two SSTables, old items can be propagated and discarded with no extra I/Os. Similarly, WiredTiger can propagate old items during checkpointing. OLCP can also be implemented in in-memory databases like Hyper [39]: Hyper maintains free lists of reusable spots, so it can propagate old items when reusing their spot (just as in our KVell+ implementation).

5 Evaluation

5.1 Goals

We evaluate OLCP queries on a variety of synthetic and production workloads. We seek to answer the following questions:

- **Resource utilization:** What is the space amplification of OLCP compared with existing SI implementations? What is the impact of using OLCP queries on throughput and tail latency?
- **Scalability:** How does OLCP scale with the number of concurrent scans and with the size of the store?
- **Performance:** How does OLCP perform on TPC and production workloads?

5.2 Experimental settings

Hardware: We use the following hardware configurations:

Config-AWS. An AWS i3.metal instance, with 36 CPUs (72 cores) running at 2.3GHz, 488GB of RAM, and 8 NVMe SSD drives of 1.9TB each (brand unknown, 2016 technology). The server can sustain a total of 3M read IOPS and 1.4M write IOPS (on read/write workloads, the maximum number of IOPS varies between 1.4 and 3M). The store is configured to cache 30GB of data.

Config-NVMe. A 4-core 4.2GHz Intel i7, 48GB of RAM, and a 480GB Intel Optane 905P (2018). The server can sustain 500K read or write IOPS. The store is configured to cache 20GB of data.

Workloads: We use the following workloads:

YCSB-T: YCSB-Transactional [21] is inspired by the Yahoo! Cloud Serving Benchmark [16] but groups updates in transactions. The average KV item size is 1024B, and the total data set size is approximately 100GB (100M keys) for the small test and 5TB (5B keys) for the large test. Similarly to previous work [77], we perform 16 updates per transaction and items are accessed uniformly. We use this workload to test the limit of SI and OLCP under a write-heavy workload (100% updates).

TPC-CH: The TPC-CH workload [15] mixes the widely popular TPC-C and TPC-H workloads. Currently, TPC-C is the industry standard to simulate OLTP systems [72] and TPC-H is the industry standard to simulate OLAP systems [74]. The TPC-CH workload harmonizes the representation of the data used by TPC-C and TPC-H so that TPC-C and TPC-H queries can run on the same dataset. The TPC-CH benchmarks [15] remove 3 updates that cause most TPC-C queries to fail due to write-write conflicts. Without this modification, TPC-C transactions abort 85% of the time. The abort rate goes down to less than 1% of the time with the modification. Our implementation is similar to the one for Redis [73].

In order to reach a significant database size, we configure TPC-CH to run with 300 warehouses. In that configuration, the store contains 140M items in total, 90M of which represent orders. The rest of the items represent customer data, stock, etc.

Production workloads from Nutanix: The production workloads are two write-intensive workloads, with a profile of 57:41:2 write:read:scan ratio. The KV item sizes range between 250B and 1KB, with a median of 400B. The total dataset size for the production workload is 256GB. The difference between the two workloads is the data skew: The key distribution in Production Workload 1 is close to uniform, while Production Workload 2 is more skewed. OLTP

transactions perform on average 10 requests per transaction.

Existing SI implementations: We compare OLCP to the conventional SI implementations and the Steam SI implementation presented in Section 4. In the remainder of this section, we refer to these implementations as "SI" and "Steam", respectively.

5.3 YCSB-T

In this experiment, we run scans of the store concurrently with YCSB-T transactions. The system is disk-bound. We run the experiment with the 100GB dataset on Config-NVMe.

5.3.1 Space amplification

Figure 4 presents the evolution of the number of old versions in time for a Zipfian and a uniform distribution of updates, varying the number of concurrent scans.

Figure 4(a) - 1 scan - Zipfian distribution: Unsurprisingly, the number of old versions increases linearly with time with the standard implementation of SI and, after 24 minutes of execution, the store has accumulated 350 million old versions and runs out of space. Steams keeps at most one version per active snapshot; since we only execute one scan, Steam only keeps at most one old version per item. Running a Zipfian workload concurrently with a single scan is the best case scenario for Steam because most updates are concentrated on a few items. At the end of the scan, Steam has accumulated 50M old versions. OLCP propagates old versions to the scan, and the number of old versions using OLCP is low and stable throughout the run (a maximum of 1000 old versions).

Figure 4(b) - 1 scan - Uniform distribution: Similarly to the Zipfian distribution, the number of old versions grows linearly with the standard implementation of SI. Because updates are distributed over more items, Steam keeps more versions and eventually the store doubles in size. The number of old versions using OLCP is again negligible throughout the run (a maximum of 1000 old versions).

Figure 4(c) - 3 scans - Uniform distribution: In this experiment, we launch a second scan after 500s of execution, and a third scan after 1,000s of execution. The three scans run concurrently. In this configuration, Steam has to keep up to three versions per item. At the end of the execution of the first scan (not shown in the picture), the store has accumulated 250M old versions (store tripled in size). OLCP propagates old versions to the scans and has close to zero space amplification (a maximum of 1000 old versions).

5.3.2 Throughput

In this section, we study the performance of the scans and the updates when executed with the various SI implementations. We run the uniform workload with a single scan presented in the previous section. Results are similar with the Zipfian distribution and with more scans. Figure 5(a) shows the scan throughput, and Figure 5(b) shows the update throughput.

Figure 5(a): The scan throughput is the same for the standard SI implementation and Steam. Surprisingly, scanning data is much faster using OLCP. The OLCP scan finishes after 691s. With standard SI, the scan aborts after 1460s because the store runs out of disk space (350GB space amplification). With Steam, the scan takes 1870s to complete, $2.7x$ as long as OLCP. OLCP queries process items just before they are overwritten, and thus when they are in memory. In contrast, with SI and Steam, queries have to fetch most of their data from disk. This advantage is especially visible at the beginning of the scan. As the scan progresses, the advantage of OLCP over SI decreases, because, statistically, as the scan progresses, most of the overwritten items have already been scanned.

Figure 5(b): OLCP scans also interfere slightly less with updates because updates make better use of the caches with OLCP. Updates happen as follows: read a 4KB block (1 I/O if the block is not cached), modify and persist the block (1 I/O). In a uniform workload, the probability of hitting the cache depends on the store size ($P(hit) = cache\ size/store\ size$). Because the database grows less with OLCP, the read has a higher probability of hitting the cache and updates are faster.

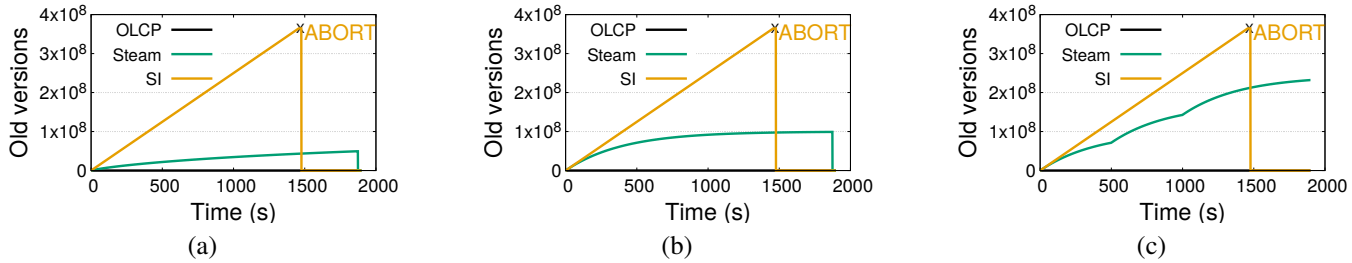


Figure 4: **Config-NVMe**. Evolution of the number of old versions for (a) a Zipfian workload with 1 scan, (b) a uniform workload with 1 scan, and (c) a uniform workload with 3 scans.

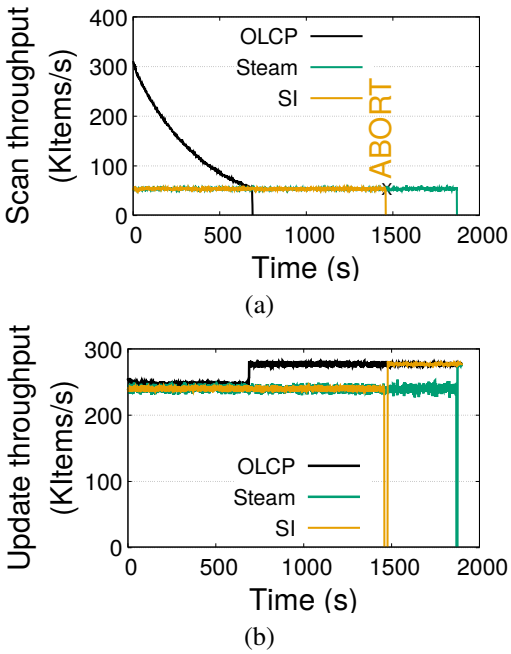


Figure 5: **Config-NVMe**. Evolution of (a) scanned items/s, and (b) updated items/s. Y-axes differ.

Table 1 shows the tail latency of the updates. At the 99th percentile, it takes 3-3.4ms to commit the 16 updates performed by an OLTP transaction (190-250us per update). Switching to OLCP for scans has no significant impact on the 99th percentile latency of OLTP transactions. In SI, the GC of the 350M old items stalls the store after the scan aborts, so the tail latency is high (18s). Cleaning the 100M old items takes 5s in Steam. A non stop-the-world GC could be used, at the expense of higher average space utilization. In OLCP, regardless of the GC implementation, cleaning overhead is negligible, and tail latency is orders of magnitude lower (9ms).

Latency	SI	Steam	OLCP
99p	3.4ms	3.1ms	3ms
Max	**18s**	**5s**	9ms

Table 1: **Config-NVMe**. Tail latency of OLTP transactions (16 updates).

Config-AWS: Trends are similar on Config-AWS. The full scan of the store finishes in 134s with OLCP and in 256s with SI and Steam. OLTP transactions have similar throughput.

5.3.3 Overhead of propagations

The overhead of propagations depends on the number of running OLCP queries: the more OLCP queries, the more enqueues in propagation queues might be done. The overhead also depends on whether concurrent updates are done on scan ranges or not: an item is only propagated if it belongs to a scan range. We launch up to 32 concurrent scans on two stores containing 100M items (100GB) and 5B items (5TB), respectively. Each scan reads a random range of 1M items. As in the previous section, the scans run concurrently with an update intensive YCSB-T workload. We run all tests on Config-AWS, since it is the only machine able to store 5TB.

Figure 6 presents the average number of scanned items per second, varying the number of concurrent scans. On all tested configurations, OLCP is equivalent or faster than conventional SI and Steam. The difference between OLCP and conventional SI is lower than in the experiments of Section 5.3.2 because (i) the queries only scan a small percentage of the store, so updates are less likely happen in a scanned range and result in a propagation, (ii) the read bandwidth of the disks on Config-AWS is higher than the write bandwidth, so the scan progresses faster than the updates. On the 100M store, the gap between OLCP and SI increases with the number of concurrent scans. Indeed, as the number of scans increases, so does the probability that a propagation happens within a scan range and that OLCP can process items in memory. On the 5TB store this effect is less visible (statistically an update has a lower probability of being in a scan range).

The throughput on the 5TB store is lower than on the 100M store because less data is cached (30% vs. 0.6%). The total number of "scans + updates" requests per second is not constant in the experiments (i.e., adding 100K scans/s does not reduce the update rate by 100K updates/s) because (i) reads are done using at most 1 I/O (vs. 2 for updates), and (ii) Config-AWS disks can sustain a higher number of read IOPS than write IOPS.

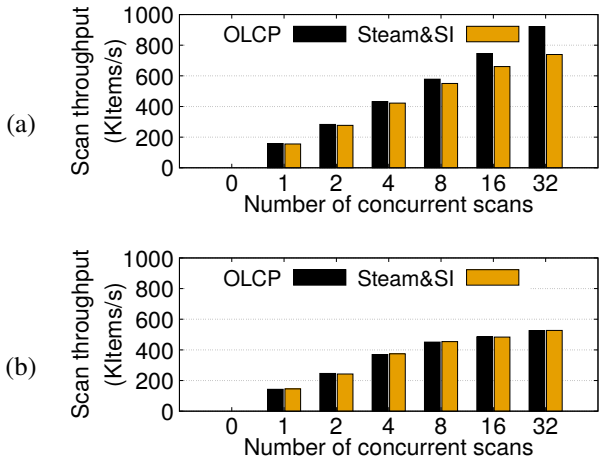


Figure 6: **Config-AWS**. Number of scanned items per second on a (a) 100M and (b) 5TB store, varying the number of scans.

Figure 7 presents the number of updates performed per second. OLCP is slightly faster for the same reasons as those presented in the previous experiment.

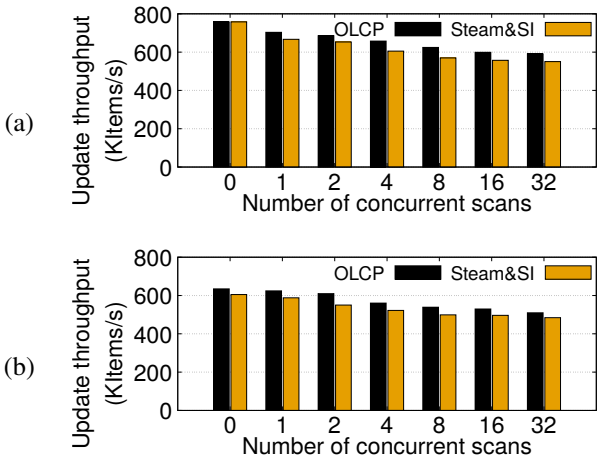


Figure 7: **Config-AWS**. Number of updates per second on a (a) 100M and (b) 5TB store, varying the number of scans.

In conclusion, it is possible to propagate items to OLCP queries with negligible overhead. In all experiments, less than 2% of the time is spent propagating values.

5.4 TPC-CH performance

In this section, we measure space amplification and performance on a TPC-C workload running concurrently with a TPC-H analytical workload. We ran on average 10 TPC-C queries concurrently. Each TPC-C query does an average of 22 requests (17 reads, 5.5 writes), and 30% of the reads hit the cache. Figure 8 presents the throughput of TPC-C running concurrently with TPC-H Query 17, presented in Algorithm 9.

With OLCP, Query 17, which scans 64% of the store, completes without space overhead and creates little interference with TPC-C queries (3% slowdown compared to an execution without a scan). Under Steam, the store doubles in size. Under SI, the store runs out of space, and the query aborts.

5.5 Production workloads performance

We study the performance of conventional SI, Steam and OLCP with the production workloads running on Config-AWS. Figure 9 presents the resulting space amplification, scan throughput and update throughput. At the end of the analytical processing, in Production Workload 2, the store has accumulated 934M old versions with the conventional SI implementation, and GC takes 49s. Steam stores 310M old versions at the end of the analytical processing. OLCP queries cause no space amplification. All implementations have the same throughput.

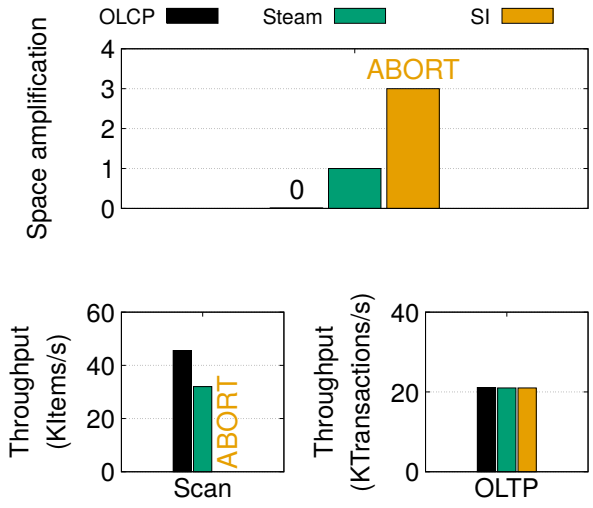


Figure 8: **Config-NVMe**. Space amplification and throughput of TPC-CH.

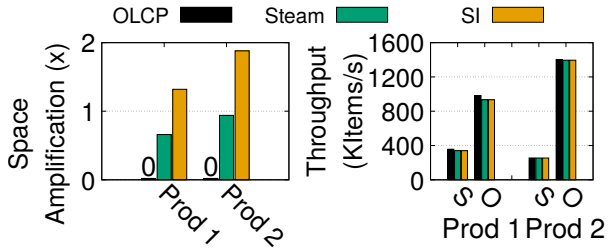


Figure 9: **Config-AWS**, production workloads. Space amplification, S (scan), O (OLTP) throughput.

6 Related Work

Running mixed OLTP/OLAP workloads: OLTP/OLAP workloads are commonly handled by systems that maintain a column-oriented datastore for OLAP (e.g., Vertica [42], C-Store [67], and Hive [71]), isolated from the row-oriented OLTP system (e.g. Cassandra [26], RocksDB [24]). This approach allows to optimize each sub-system independently. The main disadvantages are running analytics on old data and space amplification caused by data replication.

A popular approach to decrease data replication overhead is to design store for OLTP/OLAP workloads from the ground-up [10, 13, 25, 38, 39, 43, 60, 80]. Typically, these systems employ hybrid vertical/horizontal data partitioning schemes, coupled with carefully chosen secondary indexing. A significant drawback of these systems is the performance impact that OLAP and OLTP workloads have on each other (e.g., up to 5x throughput decrease in SAP HANA [63]). In OLCP, the analytics workloads do not impact transactions thanks to OLCP's minimal GC overhead.

Reducing space amplification in SI: The role of SI is to provide a coherent view of the data to OLAP queries [51, 66, 69]. SI-related space amplification is one of the most challenging issues for stores that run fully in main memory and it has been addressed by many designs. Harizopoulos et al. [28] execute transactions sequentially to avoid MVCC maintenance work. IoSnap provides flash-optimized snapshots that reduce space overhead by reconstructing snapshot metadata in-memory [68]. Hyper [38, 39] runs OLTP transactions on a `fork` of the store. BatchDB [51] runs OLAP queries on a replica of the store. These solutions still create problematic space amplification (up to $2x$), and garbage collection times at the end of the execution of OLAP queries. Furthermore, the execution of OLAP queries might be delayed to the next batch, adding possibly minutes/hours of latency to analytical queries. OLCP mitigates the space amplification and garbage collection issues. OLCP model could also be beneficial for in-memory stores.

Space amplification in KVs for fast drives: Much of the prior KVs work relies on SI to provide a consistent view of the data during range scans [2, 4, 36, 37, 47, 50, 57–59, 64, 65]. Existing systems such as PebblesDB [64], TRIAD [3], WiscKey [50], and HashKV [11] propose optimizations to decrease space amplification caused by compactions in log-structured merge KVs. SlimDB [65] decreases space for caching indexes and filters. Other KVs designed for fast drives do not support transactions [3, 5, 40, 41, 45, 48]. To the best of our knowledge, OLCP is the first work that focuses on reducing disk space amplification due to SI on fast drives.

Improving the performance of MVCC: In practice, SI is implemented through MVCC [6]. Many recent optimizations and protocols provide support for high transaction rates [8, 33, 46, 49, 55, 75]. Steam [8] trims versions that do not belong to any *active* transaction's snapshot. Steam is efficient for skewed workloads. However, under a uniform load the space amplification is proportional to the number of active transactions. We go one step further by propagating old versions to avoid keeping unnecessary versions in snapshots. Silo [75] chooses provides scalable timestamps and uses RCU to garbage collect old versions. Cicada [49] batches operations to reduce protocol costs. TicToc [77] only keeps the latest version of an item in the store. All these techniques focus on improving the speed of MVCC, but do not address space amplification. They can be used to complement OLCP.

Improving the performance of transactions: Various approaches have been proposed to increase transaction performance such as transactional memory techniques [7, 18, 22, 29, 31, 54, 62], transaction support for byte-addressable persistent memory [27, 53], work stealing [79], relaxing ACID properties when possible [17, 61, 76], decreasing replication overhead [78], and reducing coordination [70]. These techniques are orthogonal to OLCP and can be used together with our model to boost the OLTP workload.

7 Conclusion

Long OLAP queries cause problematic space amplification and long transaction tail latencies when run under SI. To remedy this problem, we propose OLCP, a new query model. OLCP provides the same isolation guarantees as conventional SI implementations, but with much reduced space amplification and interference with concurrent OLTP transactions. We show how OLCP can be used to express a wide range of OLAP queries. We implement OLCP in KVell+, an extension of KVell, a state-of-the-art open-source KV for NVMe SSDs. OLCP achieves low or no space amplification, up to 2x higher throughput for OLAP queries, and order-of-magnitude improvements in tail latency for concurrent OLTP transactions.

References

[1] Nitin Agrawal and Ashish Vulimiri. Low-latency analytics on colossal data streams with SummaryStore. In *Proceedings of SOSP*, 2017.

[2] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. BzTree: A high-performance latch-free range index for non-volatile memory. In *Proceedings of the VLDB Endowment*, 2018.

[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of USENIX ATC*, 2017.

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of USENIX ATC*, 2019.

[5] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to bε-trees and write-optimization. *;login:*, 40(5), 2015.

[6] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2), 1981.

[7] Jayaram Bobba, Neelam Goyal, Mark D Hill, Michael M Swift, and David A Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of ISCA*, 2008.

[8] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment*, 13(2), 2019.

[9] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator: Self-managing storage for enterprise clusters. In *Proceedings of NSDI*, 2017.

[10] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Es 2: A cloud data storage system for supporting both OLTP and OLAP. In *Proceedings of ICDE*, 2011.

[11] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of USENIX ATC*, 2018.

[12] Joe Chang. TPC-H SF100 Non-parallel Plans, SQL Server 2008. http://www.qdpma.com/tpch/TPCH100_Query_plans.html, 2020.

[13] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proceedings of EuroSys*, 2012.

[14] Cognos. IBM Cognos Analytics. https://www.ibm.com/products/cognos-analytics, 2020.

[15] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, 2011.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of SoCC*, 2010.

[17] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of OSDI*, 2018.

[18] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of ASPLOS*, 2006.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[21] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. YCSB+ T: Benchmarking web-scale transactional databases. In *Proceedings of ICDE Workshops*, 2014.

[22] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. Proteustm: Abstraction meets performance in transactional memory. In *Proceedings of ASPLOS*, 2016.

[23] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[24] Facebook. RocksDB: a persistent key-value store for fast storage environments. https://rocksdb.org, 2018.

[25] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database–an architecture overview. *IEEE Data Engineering Bulletin*, 35(1), 2012.

[26] Apache Software Foundation. Cassandra NoSQL key-value store. http://cassandra.apache.org/, 2018.

[27] Kaan Genç, Michael D Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of PLDI*, 2020.

[28] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018.

[29] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1), 2010.

[30] Sven Helmer, Guido Moerkotte, et al. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of the VLDB Endowment*, volume 97, 1997.

[31] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of ISCA*, 1993.

[32] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 651–665, 2019.

[33] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment*, 13(5).

[34] Hyperdex. Hyperleveldb. https://github.com/rescrv/HyperLevelDB, 2018.

[35] iccube. iccube embedded analytics. https://www.iccube.com/, 2020.

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

[37] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of USENIX ATC*, 2018.

[38] Alfons Kemper and Thomas Neumann. One size fits all, again! the architecture of the hybrid OLTP&OLAP database management system Hyper. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, 2010.

[39] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of ICDE*, 2011.

[40] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures. In *Proceedings of SC*, 2017.

[41] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *Proceedings of FAST*, 2019.

[42] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The Vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12), 2012.

[43] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment*, 8(12), 2015.

[44] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another SQL-to-MapReduce translator. In *Proceedings of ICDCS*, 2011.

[45] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of SOSP*, 2019.

[46] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High performance transactions in Deuteronomy. In *Proceedings of CIDR*, 2015.

[47] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of ICDE 2013*, 2013.

[48] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of OSDI*, 2011.

[49] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of SIGMOD*, 2017.

[50] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of FAST*, 2016.

[51] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of SIGMOD*, 2017.

[52] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing mapreduce for multicore architectures. Technical report, Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

[53] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent memory transactions (arXiv), 2018.

[54] Shuai Mu, Sebastian Angel, and Dennis Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of EuroSys*, 2019.

[55] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of SIGMOD*, 2015.

[56] Oracle. Oracle essbase. https://www.oracle.com/business-analytics/essbase.html, 2020.

[57] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of USENIX ATC*, 2016.

[58] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of SoCC*, 2018.

[59] Percona. Tokumx. https://www.percona.com/software/mongo-database/percona-tokumx, 2018.

[60] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of SIGMOD*, 2009.

[61] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of OSDI*, 2010.

[62] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of OSDI*, 2008.

[63] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Technology Conference on Performance Evaluation and Benchmarking (TPCTC)*, 2014.

[64] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of SOSP*, 2017.

[65] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of VLDB Endowment*, 10(13), 2017.

[66] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of SOSP*, 2011.

[67] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A column-oriented DBMS. In *Proceedings of the VLDB Endowment*, 2005.

[68] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a flash with IoSnap. In *Proceedings of EuroSys*, 2014.

[69] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment*, 13(2), 2019.

[70] Adriana Szekeres, Michael Whittaker, Naveen Kr. Sharma, Jialin Li, Arvind Krishnamurthy, Dan Ports, and Irene Zhang. Meerkat: Scalable replicated transactions following the zero-coordination principle. In *Proceedings of EuroSys*, 2020.

[71] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2), 2009.

[72] TPC-C. TPC-C, an on-line transaction processing benchmark. http://www.tpc.org/tpcc/, 1992.

[73] Python TPC-C. https://github.com/apavlo/py-tpcc, 2019.

[74] TPC-H. TPC-H a decision support benchmark. http://www.tpc.org/tpch/, 2018.

[75] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[76] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a distributed database. In *Proceedings of OSDI*, 2014.

[77] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of SIGMOD*, 2016.

[78] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4), 2018.

[79] Xiaozhou Zhou, Zhaoguo Wang, Rong Chen, Haibo Chen, and Jinyang Li. Extracting more intra-transaction parallelism with work stealing for oltp workloads. In *Proceedings of the Asia-Pacific Workshop on Systems*, 2017.

[80] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment*, 13(7), 2020.



Serving DNNs like Clockwork: Performance Predictability from the Bottom Up

Arpan Gujarati*
Max Planck Institute for Software Systems

Reza Karimi*
Emory University

Safya Alzayat
Max Planck Institute for Software Systems

Wei Hao
Max Planck Institute for Software Systems

Antoine Kaufmann
Max Planck Institute for Software Systems

Ymir Vigfusson
Emory University

Jonathan Mace
Max Planck Institute for Software Systems

Abstract

Machine learning inference is becoming a core building block for interactive web applications. As a result, the underlying model serving systems on which these applications depend must consistently meet low latency targets. Existing model serving architectures use well-known reactive techniques to alleviate common-case sources of latency, but cannot effectively curtail tail latency caused by unpredictable execution times. Yet the underlying execution times are not fundamentally unpredictable—on the contrary we observe that inference using Deep Neural Network (DNN) models has deterministic performance.

Here, starting with the predictable execution times of individual DNN inferences, we adopt a principled design methodology to successively build a fully distributed model serving system that achieves predictable end-to-end performance. We evaluate our implementation, Clockwork, using production trace workloads, and show that Clockwork can support thousands of models while simultaneously meeting 100 ms latency targets for 99.9999% of requests. We further demonstrate that Clockwork exploits predictable execution times to achieve tight request-level service-level objectives (SLOs) as well as a high degree of request-level performance isolation.

1 Introduction

With the proliferation of machine learning (ML), model inferences are now not only commonplace but increasingly on the critical path of web requests [29, 71]. Inference requests are handled by underlying model serving services [16, 26, 51, 58] responsible for supporting scores of different pre-trained ML models (including personalized models and experimental A/B tests), ideally at low latency, high throughput, and low cost. These are demanding goals to meet at scale—Facebook alone serves over 200 *trillion* inference requests each day [48]. Furthermore, at least 100 companies are creating hardware chips for accelerated ML inference [48], which underscores the high stakes in this industry.

* Equal contribution

Yet significant software bottlenecks continue to hamper the efficient utilization of hardware accelerators, such as GPUs, for high-performance model serving. Consider an inference request passing through a model serving system. The request has an inherent deadline after which the answer ceases to be useful to the end-user, and so the system should seek to bound the latency of the request, or even provide service level objectives (SLOs) for consistently achieving low tail latency. The canonical approach for building such a low-latency system is to reduce potential wait times for resources through over-provisioning, since a larger pool of available resources makes it more likely to find a resource on which a pending request can be immediately scheduled. Increased resource provisioning, however, comes at the expense of efficiency and utilization.

Existing systems fundamentally assume that the constituent system components have *unpredictable* latency performance [16, 58]. Moreover, the best-effort techniques employed to tolerate such variability, such as fair queuing, further cascade the unpredictability to other system components and propagate tail latency to higher layers. While some performance volatility of a model serving system is due to external factors, such as a bursty or skewed workload, much variability in execution times stems from design decisions internal to the service, ranging from caching decisions over conditional branching behavior to concurrency from other processes, the OS, and the hypervisor. The challenge, then, is to tame the internal unpredictability.

In this paper, we present the design and implementation of Clockwork, a distributed system for serving models with predictable performance. With an explicit focus on the ubiquitous deep neural network (DNNs) architectures we first show that DNN inference is fundamentally a deterministic sequence of mathematical operations that has a predictable execution time on a GPU. To leverage this observation in designing a responsive model serving system, our approach is to preserve predictability wherever possible by ***consolidating choice***: eschewing reactive and best-effort mechanisms and centralizing all resource consumption and scheduling decisions. Clockwork will only execute an inference request if it is confident that the request can meet its latency SLO. To

support such proactive scheduling, Clockwork is composed of *workers* that each handle one or more GPUs, and a centralized *controller* that schedules requests. Each Clockwork worker, responsible for the exclusive model loading and inference execution on the GPUs, achieves predictable performance. If a worker cannot execute a particular schedule, because of external factors, the request is immediately aborted and the worker resumes execution of the next request at the specified time. The Clockwork controller manages the resources of each worker and maintains a minimal advance schedule for the worker's operations, including model placement and replication.

We have implemented Clockwork in C++ and evaluated it using a wide range of DNN models on production workload traces. In comparison to Clipper [16] and INFaaS [58], two prior model serving systems, Clockwork more effectively meets latency goals while providing comparable or better goodput. Clockwork more effectively shares resources between different models, and scales to thousands of models per worker. For realistic workloads comprising unpredictable, bursty, and cold-start clients, Clockwork consistently meets low-latency response times of under 100ms.

The main contributions of this paper are as follows:

- We demonstrate that predictability is a fundamental trait of DNN inference that can be exploited to build a predictable model serving system.
- We propose a system design approach, *consolidating choice*, to preserve predictable responsiveness in a larger system comprised of components with predictable performance.
- We present the design and implementation of Clockwork, a distributed model serving system that mitigates tail latency of DNN inference from the bottom up.
- We report from an experimental evaluation on Clockwork to show that the system supports thousands of models concurrently per GPU and substantially mitigates tail latency, even while supporting tight latency SLOs. Clockwork achieves close to ideal goodput even under overload, with unpredictable and bursty workloads, and with many contending users.

2 Background and Motivation

The state of machine learning. The meteoric rise of applications driven by machine learning (ML), ranging from computer vision [28, 78] to ad-targeting [3, 17] to virtual assistants [13, 64], has prompted significant interest into making both ML training and inference faster. These efforts have targeted the underlying ML models, hardware accelerators, and software infrastructure. Chief among the ML modeling approaches are *deep neural networks* (DNNs), which are composed of multiple layers of artificial neurons tuned through non-linear convolution and pooling operations [25].

A plethora of specialized hardware are being developed and deployed for ML training and inference [48], such as ASIC and FPGA chips, Google's TPUs [41], and Facebook's Big Basin [29] chips. The dominant machine learning hardware in data centers, however, is the GPU, representing a third of the global market in 2020 [5], and will be our focus here.

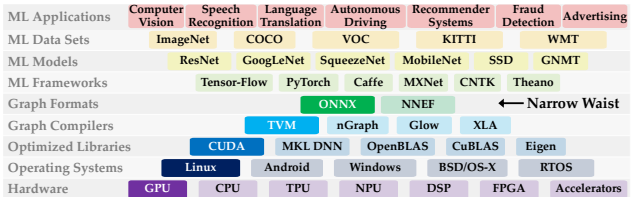


Fig. 1: Model serving targets the narrow waist of the ML software stack (adapted from Reddi *et al.* [48]). Clockwork targets the shaded blocks on the left.

Interposed between the emerging DNN applications and hardware accelerators, an ecosystem of ML software frameworks is flourishing. Fig. 1 displays several prominent projects in today's ML software stack. Layered protocol stacks in complex systems and competitive environments tend to evolve into hourglass-shaped architectures [4]. We are witnessing the ONNX and NNEF graph exchange formats for DNNs [49, 52] emerging as the "narrow waist" of the ML stack, acting as an interface between high-level ML model development and low-level software and hardware concerns.

Model serving. Operators increasingly deploy machine learning on the critical path of nascent interactive applications [71]. This has elevated machine learning inference to separate, managed *model serving* services [16, 26, 58]. From the vantage point of an operator, the model serving users (customers or internal applications) upload their pre-trained DNN ahead of time (the natural format for which is ONNX/NNEF). Their applications can then submit *inference requests* to an API. The model serving back-end manages the users' models and the hardware accelerator resources, and provides timely responses to inference requests. Upon receiving an inference request, it loads the appropriate model into hardware if not already loaded, runs the DNN on the input, and returns the resulting output to the user. Model serving has similar concerns to other datacenter services [2]: it multiplexes workloads of different users concurrently and load balances requests across multiple workers and GPU hardware accelerators.

Low-latency inference. Model serving users require a timely response to their queries. Most cloud and data center services have *service-level objectives* (SLOs) that codify the performance that clients can expect from the service [40]. The most common type is a *latency SLO*, which specifies the service's acceptable request latencies, typically on the order of milliseconds [14, 32, 41]. For example, a latency SLO might specify a 10ms average response time, or a 40ms 99th percentile response time, or both. If a service fails to meet its SLOs – for example, by being too slow for too many requests – the service provider may risk a penalty.

Model serving further operates under hard cost constraints. Specialized ML hardware is necessary to achieve interactive

latencies [41], but it is comparatively expensive to procure and operate, and must thus be used efficiently [60, 65]. Existing model serving systems achieve efficient inferences for specific heavily used models by dedicating them entire GPUs and using copious batching [41]. However, many use cases cannot justify dedicated hardware resources: applications with insufficient request volume; specialization (*e.g.* location-specific search or language-to-language translation); and experimentation (*e.g.* retrained models and A/B testing) [63]. Efficiently serving models with low request rates requires a large number of models to share accelerators; no existing model serving system supports this.

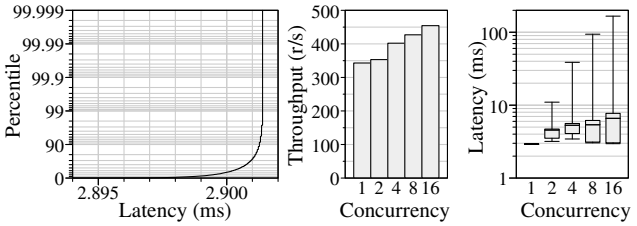
While it is already difficult for model serving operators to meet latency SLOs under these constraints, the bigger challenge lies in minimizing *tail latency*, the insidious bane of interactive performance. Numerous sources of latency variability in complex individual [46] and distributed [18, 56] systems have been identified and studied, including out-of-order scheduling, interference from concurrency, power saving modes, and network queuing delays.

The crux of tail latency lies in performance variability of both the constituent system/network components and the encompassing architecture. To tame it, the system designer can either seek to (quoting Dean and Barrosso [18]) "*create a predictably responsive whole out of less-predictable parts*", or to expend significant effort to systematically unshroud and mitigate the performance variability of these underlying components. To meet tight tail-latency SLOs under resource constraints, the latter approach is necessary.

Observation: DNN inference is predictable. We observe that DNN executions exhibit negligible latency variability, a result both intuitive in concept — DNN inferences involve no conditional branches — and demonstrable in practice. Although we describe our observations in the context of GPU execution, they extend to other accelerators such as TPUs, and also to CPU execution where appropriate.

Conceptually, a DNN inference is a fully deterministic execution. Each DNN inference request carries a fixed-size *input* tensor argument; in practical terms this is a statically-sized array of bytes. A worker receives this input over the network into main memory. To execute on a GPU, the input is copied from main memory to GPU memory over the PCIe interconnect. The DNN is then executed on the GPU. Abstractly, a DNN is a pre-defined sequence of tensor multiplications and activation functions. Concretely, the DNN code applies these operations to the input tensor one-at-a-time to transform the input into an output. DNN code lacks conditional branching; input choices such as batching size and RNN sequence length are specified ahead of time as parameters. The output is also a statically-sized array of bytes, and it is copied from GPU memory back to main memory over the PCIe interconnect.

We compiled ResNet50v2 [78] with TVM 0.7 [15] and executed 11 million inferences in isolation on a state-of-the-art NVIDIA Tesla v100 GPU using random inputs and batch size 1. We measured the latencies of each inference and show the median and high-percentile latencies in Fig. 2a. The 99.99th percentile latency was within 0.03% of the median latency.



(a) CDF of 1-thread latency (b) Inference throughput and latency. Whiskers show min and max.

Fig. 2: **Inference is predictable in isolation (left).** Running inferences concurrently gains up to 25% throughput (middle), at a cost of substantially increased latency variability (right) due to interleaved GPU and OS executions.

If DNN execution times can be measured and then accurately predicted for future inferences on that model, the next question is whether a distributed model serving system can preserve the predictable responsiveness of the core inference execution.

3 Predictable Performance

To build a responsive system through principled design, we further study the factors that can cause or amplify performance variability. Importantly, components at any level of the modern system stack can contribute to variable request latency, whether at the application layer, in the operating system, or even in the hardware [46]. Network effects and workload fluctuations add two more sources of unpredictability to distributed systems.

The whole is more than the sum of its parts. The overall system performance variability is primarily governed by how the system is assembled from its constituent components. We can handle variable latency of a software component in several ways. First, we can ignore the problem and allow the volatility to propagate to later requests or percolate to other components of the system. Even performance-conscious code that is optimized to improve throughput or average latency does not fix tail latency [19]. An example of this contagiousness of unpredictability, known as the "straggler" problem in data analytics frameworks [7, 56], is when a worker executes a request that takes unusually long and the other requests that were enqueued on the worker in the meantime then incur the extra delay from the unexpected wait-time. Ignoring the variability can further compound the problem across the system, such as when the request handler itself has variable latency [69].

Second, we can mitigate the volatility by ensuring all requests match the worst-case latency, thus exchanging lower resource utilization for predictability—often a steep price when worst-case latency is significantly higher than the median.

Third, we can minimize variability by expending more resources, again in trade for lower utilization. Some networked systems, for instance, are designed to submit the same job to mul-

tiple workers in parallel and then to cancel unneeded jobs upon successfully receiving a result from the fastest worker [18].

Fourth, upon detecting an unusual delay, we can notify a feedback mechanism to adjust the environment to lower the impact on future requests. Such "best-effort" methods are typically reactive and aimed at longer-term effects, such as by temporarily adding more resources (auto-scaling [23]), throttling requests, or balancing load.

Consolidating choice. We take a fundamentally different approach: *designing a predictable system from the bottom up*. Our strategy is to restrict the choices available to lower system layers as much as possible—a philosophy based on our observation that when executing an essentially predictable task, performance variability only arose when a lower layer in the system was given choices regarding how to execute its task. Examples from all layers of the systems stack abound, including:

- **Hardware level:** when a GPU is passed multiple CUDA kernels to execute in parallel, the GPU has the choice of how to allocate resources, including execution units and memory bandwidth, between kernels. The GPU makes these choices based on its internal state and undocumented, proprietary policies.
- **OS level**: when we create multiple threads that the operating system can execute on the same core, the OS has the choice of what threads to execute when, based on internal scheduling policies and state.
- **Application level**: when the worker processes of a distributed application each manage their own cache independently, the workers have the choice of what to cache and for how long, leading to unpredictable hit rates and latency variability [38]; similarly, when worker processes implement their own thread pools and queuing policies, they have the choice of which requests to execute first, leading to unpredictable queuing times.

Fig. 2b illustrates this: a standard design for building a worker would use thread pools serving inference requests in parallel to saturate the GPU. While concurrent threads indeed increase inference throughput by up to 25%, the factors above cause tail latency to increase by 100×.

Our approach is to *consolidate choices* in the upper layers: once a layer implements choices for lower layers based on internal state, it forces the lower layer to follow a narrow path of possible executions, causing the performance of the resulting layer to be nearly deterministic. The upper layer can then sufficiently predict the performance of the lower layers and reason with foresight about resource utilization and the anticipated execution times for all requests. The price of this strategy, however, is a tighter coupling of components and a less modular architecture.

Imperfect predictability. Notably, we can consolidate choice without requiring *perfect* predictability. Real systems will retain some unpredictable components, such as managing CPU caches or workload shifts, even after consolidating choices in its upper layers. Instead, the chief goal of concentrating these choices is to make predictable executions the common case. This frees us from implementing best-effort mechanisms to tolerate the occasional, rare instance of unpredictability; instead unpredictability can be directly treated as an error.

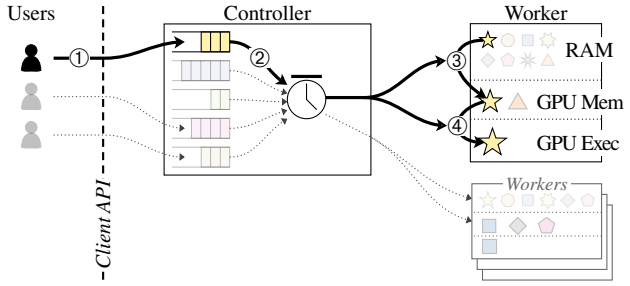


Fig. 3: Clockwork comprises multiple Workers and a centralized Controller. Models (☆) reside on Workers; inference requests are queued and scheduled centrally on Clockwork's Controller. See §4.1 for a detailed description.

4 Design

By recursively restricting choice from lower layers, we converge on a design where the most performance-critical execution choices are made in the topmost layer. In the context of a model serving service, this process converges to an architecture, which we call Clockwork, with a centralized controller and workers with predictable performance.

4.1 Overview

Architecture. Fig. 3 illustrates Clockwork's architecture. Users submit inference requests (①) which are queued centrally on Clockwork's controller. Each worker has a set of DNN models (☆) loaded into RAM and maintains exclusive control over one or more GPUs. The centralized scheduler has a global view of system state, including all workers, and decides when to execute each request (②). To execute a request, the scheduler explicitly decides when to load models into GPU memory (③) and when to execute requests on the GPU (④). At any time, the scheduler makes accurate, high-quality caching, scheduling, and load balancing decisions. The controller can perform these actions proactively because execution on workers is highly predictable. The controller transmits continual scheduling information to the workers that, by design, will execute schedules exactly as directed.

Illustrative example. To elucidate the Clockwork architectural components with more detail, including the choices that were consigned to the controller, consider the key steps for serving the inference requests illustrated in Fig. 4.

① Upon receiving an inference request r_1 for model ☆, the controller is aware that a target worker has yet to copy the model weights from RAM into GPU memory. It estimates the time required to load the model weights (LOAD), plus the time to subsequently execute the inference (INFER), and concludes that the request will complete within its specified SLO. The controller instructs the worker to copy the model weights to

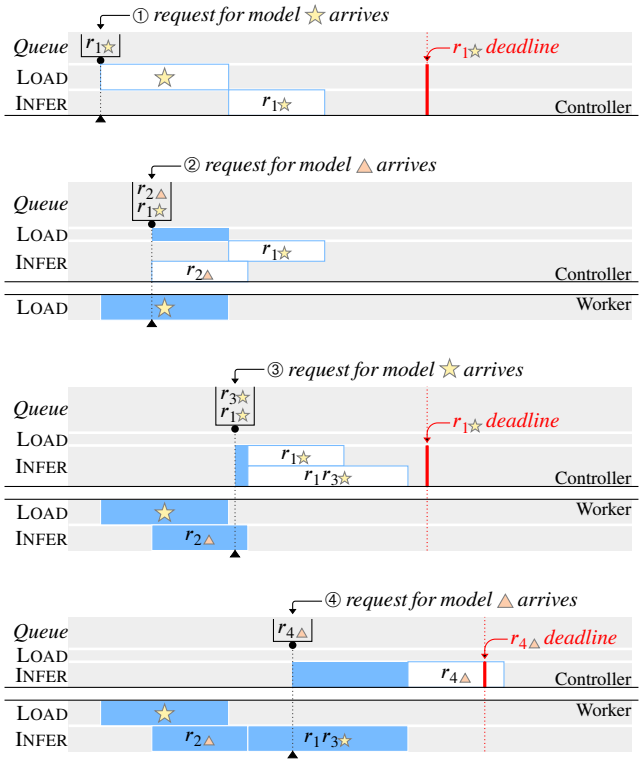


Fig. 4: Timeline of four illustrative inference requests.

GPU memory via a LOAD action. Since the controller is aware of all timings, it does not yet need to submit the subsequent INFER action until the LOAD has completed.

② While ☆ is loading, a request r_2 for model △ arrives. The controller is aware that, unlike ☆, △ is already loaded into GPU memory. The controller can choose to either INFER r_2 immediately, or wait for ☆ to complete loading then INFER r_1. Since the worker would be otherwise idle, the controller instructs the worker to execute the inference for r_2 immediately via an INFER action.

③ Clockwork workers only execute one INFER action and one LOAD action at a time, so the controller can wait until r_2 has nearly completed before submitting an INFER action for r_1. In the meantime, another request r_3 for model ☆ arrives. This gives the controller a choice between INFER for r_1 by itself, or to *batch* r_1 and r_3. Batched execution is more efficient, but takes longer. In this case a batched INFER action will still complete before r_1's deadline, so the controller instructs the worker to batch the inferences for r_1 and r_3.

④ While r_1 and r_3 execute, a request r_4 for △ arrives with a tight SLO. The controller is aware that r_4 will miss its deadline, even if it executes immediately after the worker becomes free. The controller does not proceed to schedule an INFER action, and cancels the request before performing any fruitless work.

Each step of the above execution is fast, *e.g.* for ResNet50, LOAD and INFER take approximately 8 ms and 3 ms respectively. Table 1 outlines representative measurements for 8 of the 61 models used for Clockwork experiments.

4.2 Consolidating Choice

Our design consolidates choice in three main ways. First, changes in the worker's state, for instance evicting a DNN from GPU memory, can influence the performance for future requests in a way that makes performance estimation complex. We therefore require that no worker operation should have implicit performance side-effects on any future operation. Second, we must ensure that a predictable component either delegates scheduling decisions that may impact performance to the centralized controller, or otherwise makes schedules deterministic. Third, when a predictable component is unable to execute a schedule as instructed, it is treated as an error to enable workers to get back on schedule. Workers do not attempt best-effort remediation, so as to avoid a cascade of mispredictions.

We enforce these three properties in Clockwork through an action command abstraction between the controller and workers that, in lieu of traditional RPC calls, either communicates a change in a worker's state or a task for a worker to execute. Each action the controller issues to a worker, such as LOAD and INFER, has predicted execution time and a designated execution window. These are derived using the known state of the worker, previously submitted actions, and known transitions in controller-maintained worker state.

4.3 Challenges for Predictable Inference

To consolidate choice we must first identify where performance-critical choices arise in system components. We have established that DNN inference itself on a GPU has deterministic performance; we next study the challenges in extending this result to a full-fledged inference system.

Managed memory and caches can be unpredictable (C1). RAM and GPU memory on a worker constitute state that impacts the performance of future requests. Additionally, some memory allocators exhibit variable timing for allocation and deallocation requests due to internal trade-offs between memory fragmentation and amortized performance. Memory that is used as a cache specifically introduces performance variability between cache hits and misses, with an internal cache replacement policy influencing performance of future items. To maintain predictability, we must instead consolidate choice by managing cache admission and eviction for each worker at the central controller. Fortunately, caching of DNN weights is coarse-grained and per-model.

Hardware interactions can be unpredictable (C2). Many system resources are implicitly administered by hardware schedulers that operate at very fine time-scales and produce different schedules under even minute shifts in the arrival times of other requests. The volatility of timing coupled with proprietary and un-documented scheduling policies make it onerous to accurately predict completion times for concurrent requests. The remedy for non-determinism is to strip away the ability for schedulers to reorder requests by forcing only a single request to be executed at a time, at the cost of spending

Model Family	Model	IO Size (kB)		Weights		GPU Execution Latency (ms)				
		Input	Output	Size (MB)	Transfer (ms)	B1	B2	B4	B8	B16
DenseNet [36]	densenet169	602	4	56.5	4.50	5.18	6.29	8.57	12.82	21.85
Inception v3 [68]	inceptionv3	1073	4	95.3	7.77	4.46	6.85	10.99	16.45	26.17
Mobile Pose [72]	mobile_pose_mobilenetv3	590	209	19.0	1.55	1.29	1.92	3.13	5.71	11.62
ResNet [30]	resnet18	602	4	46.7	3.81	1.27	1.86	2.73	4.06	7.02
	resnet50	602	4	102.3	8.33	2.61	3.78	5.61	9.13	15.67
	resnet152	602	4	240.9	19.58	7.71	11.14	16.21	26.48	44.60

Table 1: Measurements of a representative subset of the 61 models used for Clockwork experiments. Pre-trained models were sourced from the ONNX Model Zoo [53] and the GluonCV Model Zoo [28], and optimized for NVIDIA Tesla v100 GPUs using TVM v0.7 [15].

greater effort on keeping the resource fully utilized. Mercifully, one-at-a-time execution of DNN inferences on GPUs has closely comparable throughput to concurrent execution (Fig. 2b) and many classes of DNNs (*e.g.* convolutional neural networks) can saturate GPUs with small batch sizes.

External factors can trigger performance variance (C3). Even after systematically removing the key internal sources of unpredictability by consolidating choice, there will always remain external sources outside of the controller's purview. These include performance interference through shared network bottlenecks, thermal throttling of CPUs and GPUs, and others. The only option is to minimize their effects by building sufficient tolerance into the system.

4.4 Predictable DNN Worker

At a high-level, Clockwork workers maintain DNNs in memory and execute inference requests on one or more GPUs. The workers interface with the controller to receive actions.

Memory management. Model weights must be present in GPU memory to execute an inference. However, GPU memory capacity is small (≤32GB) relative to host memory (≤4TB), and host-to-GPU memory transfers (≈8.3ms for ResNet50) typically take longer than running the DNN inference on the GPU (≈2.9 ms). Consequently, Clockwork treats GPU memory as a cache, letting commonly or recently used models avoid expensive loads. To overcome **C1**, workers explicitly expose LOAD and UNLOAD actions to the controller for copying models to and removing models from worker's GPU memory with deterministic latency. These actions also update the state that the controller tracks for the worker.

Inference execution. The controller only sends an INFER action when a model is present in GPU memory or a LOAD action will momentarily complete. The worker internally divides INFER actions into three steps. First, INPUT transfers the input vector from host to GPU memory. Next, EXEC performs the actual heavy-weight DNN GPU calculations, which dominate the total inference time. Finally, OUTPUT transfers the resulting output vector from the GPU back to host memory. These steps may coincide: the previous request's outputs can be copied at the same time as the current request's input is being transferred. However, multiple concurrent EXEC calls cause the GPU hardware scheduler to behave unpredictably (**C2**). Fortunately, a DNN inference call by itself can efficiently utilize the GPU while also restricting the hardware scheduler to a single, predictable option (Fig. 2b). Clockwork workers therefore run a single EXEC at a time, a design choice that reduces performance variability by two orders of magnitude while only minimally decreasing inference throughput (Fig. 2b).

Interface with the controller. Clockwork workers receive LOAD, UNLOAD, and INFER actions from the controller with detailed timing expectations attached:

`type`	INFER, LOAD, or UNLOAD
`earliest`	the time when this action may begin executing
`latest`	when this action will be rejected

Rather than executing actions in a work-conserving, best-effort manner, workers strictly follow the schedule of actions imposed by the controller. The controller communicates two timestamps with every action, `earliest` and `latest`, to designate a time interval during which the worker may begin executing the action. Actions that cannot start within the prescribed window are cancelled and never executed. This allows workers to quickly get back on schedule after an individual action is delayed unexpectedly (**C3**) by skipping one or more actions, minimizing the impact of the delay on other actions. Workers communicate the result of each action back to the controller, including whether the command was successful and the measured execution time.

4.5 Central Controller

All decision-making in Clockwork occurs in the central controller. The controller receives inference requests from users and decides worker actions while striving to meet SLOs.

Modeling worker performance. The controller maintains a per-worker, per-model performance profile comprising processing time measurements of recent requests; profiles are updated continuously to tolerate shifts due to external factors (**C3**). The controller also tracks the outstanding actions and memory state at every worker. Since actions have inherently deterministic latency by design, the controller can deduce the earliest time that a worker could begin executing a new action (queuing time).

Action scheduler. The Clockwork controller proactively manages action schedules for workers. It utilizes a global view

of system requests, up-to-date worker performance profiles, and accurate predictions for when outstanding actions will complete. The controller attempts to pack worker schedules tightly by making narrow, realistic estimates for the `earliest` and `latest` time interval. The interval width balances a trade-off between Clockwork SLO fulfillment and system goodput. On one hand, making the interval too narrow increases the risk of an action not being executed by a worker because it could not be completed in time (**C3**), potentially triggering an SLO violation. On the other hand, underestimating the window length can create periods of inactivity and decrease worker utilization, thus affecting Clockwork goodput.

The scheduler lazily decides which worker should execute the inference. The controller only submits a minimal amount of work to keep workers utilized; it is in no hurry to commit because it can accurately predict action timings. Delaying choices on the controller improves schedules by providing more options, permitting the Clockwork controller to re-order and *batch* inference requests to the same model, significantly improving resource efficiency and throughput.

In our design, any worker can process any request since they all store every model in host memory; however, workers have different sets of models loaded into their GPU memory. A worker that executes only cold inferences must transfer weights for each model from host memory to the GPU and may saturate the available PCIe bandwidth, whereas a worker that executes only hot inferences may be bottlenecked by the GPU. The Clockwork scheduler balances load by mixing and matching hot and cold inferences among all workers.

5 Implementation

Clockwork's implementation, comprising 26KLOC of C++, contains various decisions that enable Clockwork to consolidate choice on its controller.

5.1 Models

Predictable model execution. Prior model serving systems such as Clipper [16] and INFaaS [58] act as orchestration layers atop existing model execution frameworks such as TensorFlow [1] and TensorRT [50]. This decoupling makes it difficult to consolidate choice, since the model execution frameworks encapsulate scheduling and memory management decisions that we wish to make with Clockwork. Instead, Clockwork implements its own model runtime, reusing key components of the TVM optimizing compiler [15]. Clockwork's model runtime enables fine-grained control over each stage of a model's execution. For models provided to Clockwork (*e.g.* in ONNX form), we compile a binary representation using TVM and postprocess the model to produce the following:

- **Weights:** A model's weights are a binary blob (10s to 100s of MB (cf. Table 1).
- **Kernels:** The CUDA kernels that execute a model (10s to 100s of kB). These are not provided by the user; they are derived from the abstract model definition, and kernels from different users can safely execute within the same process. Clockwork uses the kernels compiled by TVM. Clockwork compiles kernels for multiple configurable batch sizes; by default 1,2,4,8, and 16. Kernels for different batch sizes can use the same weights without modification.
- **Memory metadata:** At runtime, models do not directly allocate memory; instead, Clockwork will pre-allocate and manage all GPU memory and pass pointers as arguments to function calls. The memory requirements for a model are static, and Clockwork precalculates the required workspace memory and offsets required for each kernel.
- **Profiling data:** Clockwork runs a brief profiling step to produce a seed estimate for model execution times.

Model loading. Models are stored in an efficient serialized form on disk. Clockwork workers pre-load models from disk into main memory on worker startup. For the worker machines used in our evaluation, 768GB RAM can support thousands of models (cf. §6.5). Once a model is in main memory, Clockwork extracts and links the CUDA modules needed for its execution. To improve predictability, Clockwork disables JIT compilation and the caching of CUDA kernels.

5.2 DNN Workers

Each machine runs one worker process that receives and executes actions from Clockwork's controller. We do not run Clockwork in a container or VM to avoid the performance interference such sharing can impose.

Managing model weights in memory. Clockwork pre-allocates all GPU memory and divides it into three categories:

- **Workspace:** Models require a variable amount of GPU memory for intermediate results. This memory is transient and only needed during execution; once an output has been produced, it is no longer needed. Clockwork only executes models one-at-a-time, so it allocates 512MB workspace memory.
- **IOCache:** Although Clockwork only executes models one-at-a-time, Clockwork asynchronously copies inputs to the GPU prior to execution, and outputs to host memory after execution. Clockwork allocates 512MB device memory for temporary storage of inputs and outputs before and after execution.
- **PageCache:** The remaining device memory is used for storing model weights, divided into 16MB *pages*. Multiple tensors can occupy the same 16MB page and the mapping of tensors to pages is determined statically at model-compile time. At runtime, page pointers are passed as kernel arguments and tensors are read from pre-defined offsets.

Clockwork's PageCache has several advantages. First, avoiding repeated memory allocation calls leads to more predictable executions, since memory allocation can be an unpredictable source of overheads (**C1**). Second, paging *simplifies choice*: external memory fragmentation issues are

eliminated, and the controller need only track the number of total free pages to completely capture the worker's memory state. Paging slightly increases memory utilization; however, model memory requirements are static and known ahead of time, and can be bucketed on to pages to reduce internal fragmentation. Paging does not affect the latency of memory transfers.

Actions. To orchestrate workers, the controller uses the previously described *action* abstraction. Actions contain a unique `id` and an action-dependent `payload` (*e.g.* INFER inputs). Each worker runs a dedicated *executor* for each action type and each worker-GPU. An executor runs a thread that dequeues actions chronologically by `earliest` timestamp, and waits until `earliest` is reached before proceeding with an action. Executors reject actions whose `latest` timestamp has passed. To reduce interference between threads and other processes, each executor is pinned to a dedicated core and runs at real-time priority. Both INFER and LOAD execute asynchronous work in their own CUDA streams. Each executor is bottlenecked by a different resource (*e.g.* GPU execution and PCIe transfers) and can run concurrently with negligible interference.

Results. A network thread maintains a persistent connection with the controller for receiving actions and sending results. A result comprises the following:

`status`	success or an error code
`timing`	start and end times, and on-device execution duration for any asynchronous work

LOAD actions acquire pages from the PageCache, then copy weights to those pages. If no pages are available then LOAD aborts. The controller explicitly frees pages with UNLOAD; this only updates in-memory metadata and always succeeds.

INFER actions comprise INPUT, EXEC and OUTPUT, each of which have dedicated executors. INPUT executes immediately on receipt of INFER; it acquires IO memory from the IOCache then copies inputs. EXEC inherits the INFER action's `earliest` and `latest` timestamps; it checks weights and inputs are present then executes kernels on the GPU, using Workspace for intermediate calculations. OUTPUT immediately copies outputs back to main memory then releases the IO memory. To simplify controller decision making, INPUT and OUTPUT are not exposed as actions since they are orders of magnitude faster than EXEC and LOAD (10s of microseconds) for our workloads. Clockwork's memory management allows for back-to-back INFER actions for the same model.

5.3 Central Controller

On startup, Clockwork's controller establishes persistent connections to all workers and exchanges metadata about the size of each worker's PageCache, the models present on each worker, and their initial pre-profiled execution times. The core duty of the controller is to satisfy requests received from clients by submitting actions to workers. This decision making is encapsulated in the *Scheduler* interface:

`onRequest`	client request received, specifying a model ID, SLO, and providing inference inputs
`onResult`	a result is received from a worker

A scheduler implements this interface, and can invoke `sendAction` to send an action to a worker, and `sendResponse` to respond to a client. A separate layer of the controller implements common tasks such as networking, forwarding inputs to workers, setting timestamps, and handling timeouts. This design concentrates all choice in a single place, and enables different scheduler implementations to be easily dropped in.

Managing worker state. The controller maintains an accurate representation of workers' execution state, which is threefold: *memory state*, in which the scheduler tracks what models are present in the worker PageCaches and when LOAD will be required; *action profiles*, which are measurements of past 10 actions duration, stratified by model, worker, and batch size, to predict the duration of future action; and *pending actions*, which tracks submitted actions and estimates when each executor will next be available. Taken together, these enable the scheduler to accurately predict when candidate actions will complete, and avoid submitting work that cannot complete before the request's deadline. Worker state is not a significant scalability bottleneck; action profiles require only 40 bytes for each model, worker and batch size combination.

Scheduling INFER. Upon arrival, requests are enqueued into per-model request queues. For each INFER executor, a new action must be scheduled whenever the executor has less than 5 ms of outstanding work. To schedule an INFER action, a model and batch size must be selected. The batch size can differ action-to-action, though the scheduler prioritizes larger batch sizes for efficiency.

At any point in time, a model will have zero or more queued requests. However, not every request is suitable for every batch size. Higher batch sizes take longer to execute, so a request close to its deadline might only be satisfiable using a small batch size. To handle this, each model has a request queue per batch size (we term this a *batch queue*). New requests are enqueued into *every* batch queue. Requests are dropped from batch queues when they cease to be satisfiable; *e.g.* a request in the batch size of 16 queue will be dropped sooner than it is dropped from the batch size of 8 queue.

To decide which model and batch size to schedule, we use *strategies*. A strategy specifies a *model*, a *latest* timestamp, and a *batch size*. Each INFER executor has a separate strategy queue, ordered by *latest*, containing only strategies for models it has loaded. The scheduler dequeues strategies until it finds one that is *valid*: *latest* has not elapsed, and the batch queue for the specified batch size has sufficient requests. If a strategy is valid, the scheduler will also speculatively increase the batch size as long as extra requests are available.

When a valid strategy is found, an INFER action is created and requests are dequeued to fill the batch. Old strategies for this model are removed from the strategy queue, and new

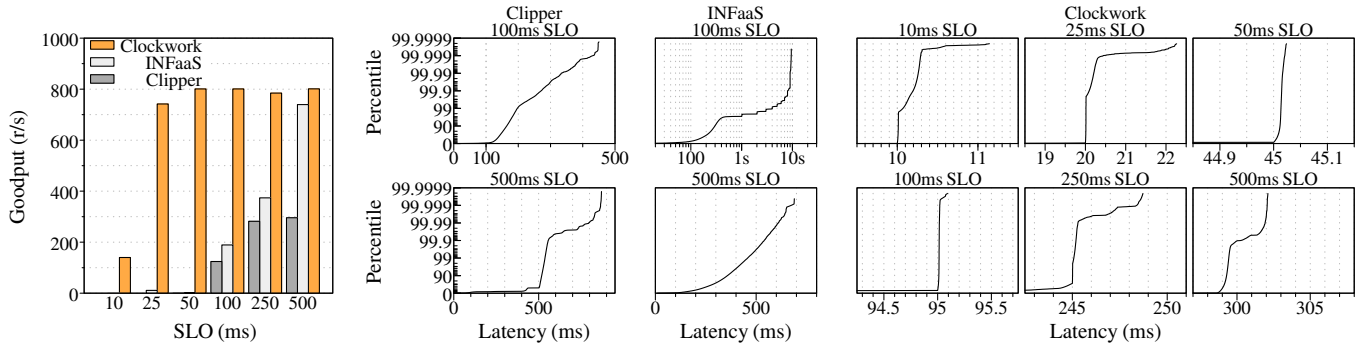


Fig. 5: Goodput and latency measurements for Clipper, INFaaS, and Clockwork. We deploy 15 instances of ResNet50 on 1 worker; each model submits 16 concurrent requests in a closed loop. (Left) Request goodput. Goodput only counts requests that succeed within the SLO. (Right) Request latency CDFs across all requests (including those rejected due to missed deadlines). Latency CDFs are scaled to highlight tail latency.

strategies are then created and enqueued. A strategy is created per batch queue; *latest* is calculated by subtracting the batch execution time from the deadline of the request at the head of the queue. Empty batch queues are skipped.

Scheduling LOAD. Each LOAD executor also schedules up to 5 ms of outstanding work. For a LOAD executor, the scheduler selects a model by estimating each model's SLO violations given the model's current state and outstanding requests. To do this efficiently, the scheduler maintains and incrementally updates *load* and *demand* statistics for models and GPUs:

- d_m the total demand for each model m
- $a_{m,g}$ the demand allocation of model m on GPU g.
- $\ell_g = \sum_m a_{m,g}$ the total load on each GPU g

A model's total demand d_m is the total estimated execution time of m's outstanding requests; we update d_m when requests for that model arrive and complete. The demand allocations $a_{m,g}$ for m on GPU g are also updated when requests arrive and complete; they are calculated such that $\sum_g a_{m,g} = d_m$. Demand allocations are 0 for GPUs where the model is not loaded. On GPUs where the model is loaded, demand allocations are inversely proportional to the GPU's load, since overloaded GPUs will be able to execute proportionally less of the total demand. Each GPU's total load ℓ_g is the sum of its allocations across all models. With these estimates, each model's load priority is defined as

$$p_{m,g} = d_m - \sum_g a_{m,g} \cdot \frac{\text{capacity}_g}{\ell_g}.$$

A model's load priority estimates its unfulfilled work. For example, a model that is not loaded on any GPUs has priority equal to its outstanding work; a model loaded on a GPU that sits mostly idle has negative priority since the GPU can serve more work than the model demands.

Clockwork does not attempt to converge to a perfect demand allocation each time the system's state changes. Rather, Clockwork incrementally updates each model's demand allocation and load priority **(i)** when new requests arrive for that model; **(ii)** when an INFER is initiated for that model; **(iii)** when LOAD and UNLOAD affect a model; and **(iv)** when a request crosses the point where it can benefit from LOAD before its deadline.

The scheduler selects LOAD actions by choosing the highest priority model that is not already loaded. Notably, models with negative priority need not be loaded since their demands are already met. Clockwork uses a least-recently-used (LRU) eviction policy when selecting models to UNLOAD.

6 Evaluation

We next assess Clockwork's ability to reliably serve DNNs under a variety of workload conditions. We begin our experimental evaluation with simple workloads in controlled settings, before expanding to heterogeneous models and diverse workloads. Our evaluation shows that Clockwork's assumptions about predictability hold, and result in a system that can effectively meet SLOs and drastically reduce tail latency.

Experimental setup. We deploy Clockwork in a private cluster of 12 Dell PowerEdge R740 Servers. Each server has 32 cores, 768 GB RAM, and 2×NVIDIA Tesla v100 GPUS with 32 GB memory. The servers are connected by 2×10 Gbps Ethernet on a shared network. In all experiments, we run the controller, clients, and workers on separate machines.

6.1 How Does Clockwork Compare?

We begin with a comparison to two prior model serving systems, Clipper [16] and INFaaS [58]. For Clipper and Clockwork, we provision a single cluster machine to use 1 GPU to serve 15 separate copies of ResNet50. ResNet50 is the *de facto* model used for comparison previously by these systems; we chose 15 models as this reached the memory limit of Clipper[1]. To evaluate INFaaS, we deployed an `m5.24xlarge` and a `p3.2xlarge` EC2 instance as the master and the worker, respectively. These are not identical experiment conditions; however, INFaaS is tightly integrated with EC2, and could not be deployed on our cluster infrastructure. We include these results for qualitative comparison.

Offered load. For each model, we run 16 closed-loop clients[2]. The serving systems may batch requests for the same model instance, but requests to different instances cannot be

[1] INFaaS memory limits were reached at 64 models
[2] Open-loop clients yielded similar results

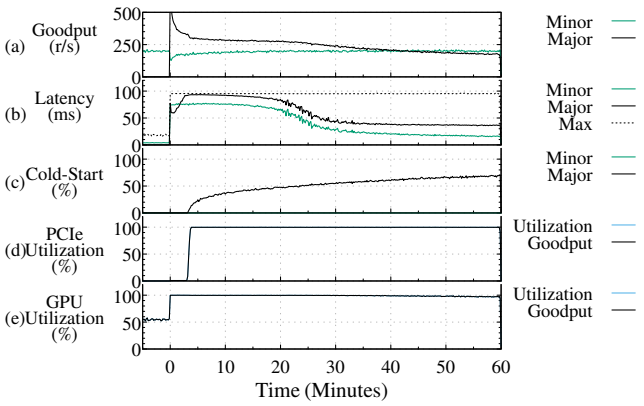


Fig. 6: Clockwork can serve thousands of models from a single worker. From $t = 0$, the Major workload adds an additional model per second, to a total of 3,600 models at $t = 60$ (cf. §6.2.)

batched. We run multiple experiments, varying the target SLO from 10 ms to 500 ms.

Goodput. Fig. 5 plots the *goodput* achieved by each system as the target SLO varies from 10 ms to 500 ms. Goodput is the number of successful requests that completed within the target SLO; it excludes timed out requests and requests that responded after the SLO.

With a high SLO of 500 ms, Clockwork and INFaaS meet their SLOs and have comparable goodput of approximately 800 r/s. Clipper's goodput is substantially lower, as Clipper only treats SLOs as an average latency target, not a strict threshold, and converges to this target over time without bounding latency variability. As SLOs tighten, goodput and tail latency deteriorate for both Clipper and INFaaS, and their goodput collapses below a 100 ms SLO. Like Clipper, INFaaS uses the SLO as a coarse-grained goal for reactive policies. Consequently, only Clockwork can continue serving SLOs below 100 ms.

Fig. 5 also plots latency CDFs for Clipper, INFaaS, and Clockwork. We scale the CDFs to emphasize tail latency. The figure illustrates how both Clipper and INFaaS allow latency higher than their SLOs. However, of note, with a 500 ms SLO, INFaaS successfully finds a configuration that can serve this SLO, and meets its SLO for 99% of its requests. By comparison, Clockwork's tail latency remains very close to the SLO in all cases. For the 500 ms SLO, Clockwork's latency remains at ≈300 ms because it schedules each model's entire batch of 16 requests at a time, round-robin across models. With 15 models and a 20 ms batch-16 execution duration, Clockwork does not exceed the optimal 300 ms latency.

6.2 Can Clockwork Serve Thousands?

The previous experiment represented an idealized scenario, with only a small number of models, each with a steady sustained workload. We now examine the serving limits of a single worker. We deploy 3,601 copies of ResNet50 to a worker, and set a 100 ms SLO. We submit two workloads: a Major workload and a Minor workload. The Major workload comprises 3,600 model instances; we vary the number of instances that are active at any point in time, and evenly distribute a workload of 1,000 r/s across all active models. The Minor workload is a single model instance that maintains a fixed 200 r/s request rate throughout the experiment.

Figure Fig. 6 (a) plots the goodput achieved by the major and minor workloads. From t=−5 to t=0 (we denote t in minutes) only the Minor workload is present, achieving its full 200 r/s. At t=0, we activate one model instance of the Major workload; the addition of 1000 r/s fully saturates the GPU (e). After that, we activate an additional model of the Major workload every 1 second. As more model instances become active, the Major workload's goodput drops since each additional model forgoes batching opportunities. At t=60 all 3,600 models are active, each submitting approximately 0.28 r/s.

By t=3.5, 201 models have been activated, reaching the capacity of GPU device memory. To continue serving requests, Clockwork begins swapping models on and off GPU; Fig. 6 (d) shows PCIe utilization rapidly rises to 100%. As more models activate, an increasing number of requests in the Major workload find that their model is not loaded; Fig. 6 (c) plots the rise in *cold-starts*, reaching 70% by the end of the experiment. The minor workload, with its sustained request rate of 200 r/s, does not experience any cold starts because its demand dwarfs every other model after the first 5 seconds. As the number of cold-starts increases, the demand on GPU execution decreases, enabling the Minor workload's goodput to gradually grow back to 200 r/s. At approximately t=20, the bottleneck for the Major workload shifts to PCIe utilization, enabling the Minor workload's latency to drop back to an average of 20 ms (b).

This experiment illustrates how bottlenecks in Clockwork can shift as workload demand changes. Clockwork can deal with shifting bottlenecks even while serving a large number of models. As illustrated in Fig. 6 (b), the maximum request latency across the experiment did not exceed the 100 ms SLO.

6.3 How Low Can Clockwork Go?

Clockwork's predictability and centralized decision-making enables it to satisfy low-latency SLOs. In this experiment, we use six Clockwork workers and evaluate the lower limit on SLOs that Clockwork can achieve by measuring the proportion of successful requests while varying the SLO. We repeat the experiment for six different workloads, varying the number of ResNet50 instances ($N = 12$ or 48) and cumulative request rate ($R = 600$ r/s, 1200 r/s, or 2400 r/s). For each experiment run, we begin with an SLO of 2.9 ms (1× the execution latency of batch-1 ResNet50 inference). Every 30 seconds, we extend the SLO by 50%; by the end of the experiment the SLO reaches 74ms. We run a separate open-loop client for each model with a Poisson inter-arrival time distribution, and as before, all models are independent (requests cannot be batched across models).

Workload satisfaction. Fig. 7 plots the *workload satisfaction* for each experiment run. Workload satisfaction is the ratio of goodput to offered load. A workload satisfaction of

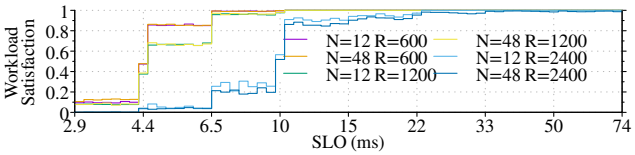


Fig. 7: Workload satisfaction rates as we vary N, the number of clients, and R, the request rate.

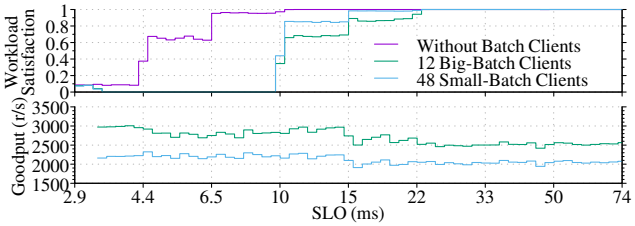


Fig. 8: Workload satisfaction rates for latency-sensitive clients (top) and workload goodput for batch clients (bottom).

1 means all requests received a successful response within their SLO. For a load of R=600 r/s and 1200 r/s, irrespective of the number of models, Clockwork successfully satisfied tight SLOs 10 and 22 ms. Even at R =2400 r/s, Clockwork comfortably managed an SLO of 74ms.

6.4 Can Clockwork Isolate Performance?

Clockwork can satisfy tight SLOs for latency-sensitive clients in isolation; we next consider when the system is shared with other users serving batch requests without latency SLOs. As before, we use six Clockwork workers, and all clients use instances of ResNet50. We provision six *latency-sensitive* clients, each submitting a 200 r/s open-loop workload. We also provision several *batch clients*, which submit sustained closed-loop workloads and do not have latency SLOs. *Big-batch* clients have a concurrency of 16, while *small-batch* clients have a concurrency of 4. Varying the concurrency affects the maximum batch size Clockwork can achieve for batch client requests. We considered three scenarios: **(a)** baseline without batch clients; **(b)** 12 big-batch clients; and **(c)** 48 small-batch clients

Fig. 8 illustrates the workload satisfaction rates for latency-sensitive clients and the total goodput achieved for the batch clients. Clockwork successfully prioritizes latency-sensitive requests over batch requests. Through SLO-aware scheduling, it ensures that the workload satisfaction rates are unaffected by the presence of other pending, less time-critical requests. At the same time, Clockwork does not throttle batch requests entirely, but schedules them during idle times or expected idle times. However, when the SLOs are too tight (<15ms), many latency-sensitive requests are rejected in advance, allowing pending batch requests to pass through.

6.5 Are Realistic Workloads Predictable?

We now ask whether executions remain predictable under realistic workloads that comprise many concurrent users and models. We also investigate whether Clockwork effectively exploits this predictability.

Model Family	Count	Model Variants
DenseNet [36]	4	121, 161, 198, 201
DLA [75]	1	34
GoogLeNet [67]	1	
Inception [68]	1	v3
Xception [68]	1	
MobilePose [33]	4	SPRN18, MNv3, RN18, RN50
ResNeSt [78]	4	14, 26, 40, 101
ResNet [30]	22	18, 18b, 34, 34b, 50, 50b, 50c, 50d, 50s, 50-1.8x, 101, 101b, 101c, 101d, 101s, 101-1.9x, 101-2.2x, 152, 152b,152c, 152d, 152s
ResNet-v2 [31]	5	18, 34, 50, 101, 152
ResNeXt [73]	3	50-32, 101-32, 101-64
SENet [35]	2	50-32, 101-32
TSN [70]	7	iv1, iv3, r18, r34, r50, r101, r152
Wide ResNet [76]	3	16-10, 28-10, 40-8
Winograd [45]	3	RN18, RN50, RN101

Table 2: List of models used in experiments.

To answer these questions, we deploy Clockwork on 12 workers and replay a workload trace of Microsoft Azure Functions (MAF) [61]. The trace records approximately 46,000 function workloads, counting the number of invocations of each function, every minute, for two weeks. It interleaves a wide range of workloads, including heavy sustained workloads, low utilization cold workloads, bursty workloads that fluctuate over time, and workloads with periodic spikes [61]. We believe this to be a representative workload for evaluation since serverless platforms enable a wide range of applications and supporting ML inference on serverless is an active area of research [10, 39].

In this experiment, we replay six hours of the MAF trace in real-time. We use 61 different models (Table 2) taken from the ONNX Model Zoo [53] and the GluonCV Model Zoo [28]. We duplicate each model 66 times, resulting in a total of 4,026 instances and reaching the main-memory capacity of our worker machines. We replay ten or eleven function workloads for each model instance. We configure Clockwork with a 100 ms SLO.

Clockwork with realistic workloads. The time series in Fig. 9 (a) shows the offered load and goodput achieved across all models. For the 6 hour experiment, both the offered load and goodput averaged 9,638 r/s – out of a total of 208 million requests, only 58 failed due to action timing mispredictions, and no requests timed out. All GPUs were fully utilized throughout the experiment, yet no request exceeded the 100ms SLO.

Fig. 9 (b) plots the median, 99th percentile, and maximum request latency over the course of the experiment. Latency spikes occur every 5, 15, and 60 minutes, due to the presence of numerous periodic workloads within the trace [61]. Workload spikes do not cause SLO violations because of latency headroom; Fig. 9 (c) shows the average batch size for the experiment, and with each workload spike, Clockwork can schedule larger batches, with higher latency. To evaluate the cold-start behavior of this workload, we categorize a request as a cold-start if its model is not already loaded into GPU's memory before arrival. For each 1-minute interval, Fig. 9 (d) counts the number of

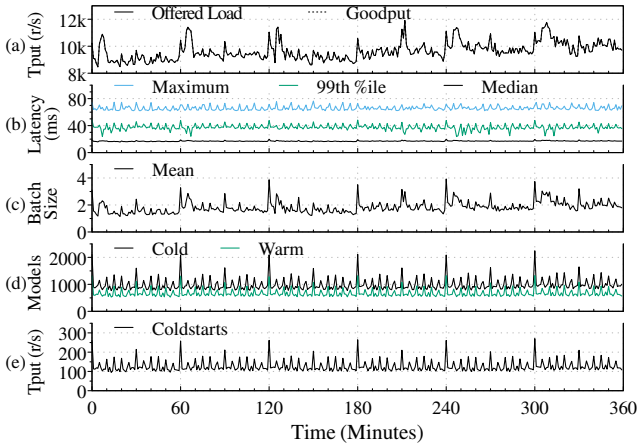


Fig. 9: Microsoft Azure Functions (MAF) over Clockwork; see §6.5 for a description.

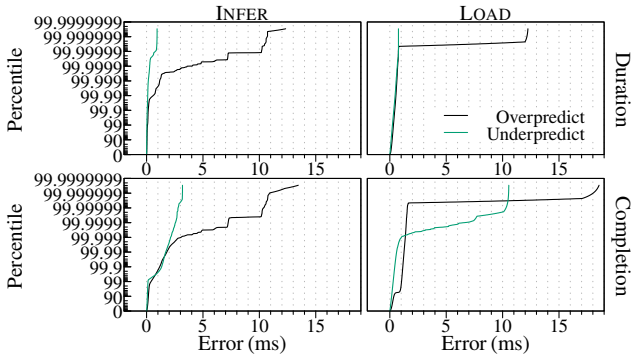


Fig. 10: Clockwork prediction and completion errors for MAF trace.

unique models that have at least one cold-start, and at least one warm-start. On average, 987 unique models perform cold-starts each minute; or approximately 25% of all models. However, while many models perform cold-starts, they only represent a small fraction of all requests. Fig. 9 (e) plots the throughput of cold-start requests, averaging 126 r/s, or 1.3% of all requests. These results show that Clockwork can sustain significant load for varied, realistic workloads comprising thousands of models.

Predictable executions. Clockwork's scheduler relies on accurate predictions of action latency, so to assess Clockwork's underlying assumptions of predictability, we next evaluate the accuracy of Clockwork's predictions. We measure the latency of INFER and LOAD actions on Clockwork's workers and compare it to the time estimated by Clockwork's controller to derive a *prediction error*. Prediction errors comprise two types: *overprediction*, when the real execution latency is faster than predicted; and *underprediction*, when the real execution latency is slower than predicted. Consistent overpredictions can lead to idle resources, while consistent underpredictions can cause SLO violations. Fig. 10 (top) plots the prediction errors for INFER and LOAD actions. For INFER actions, the 99th percentile of overpredictions and underpredictions is 144 μs and 55 μs, respectively. Thereafter, the tail latency grows

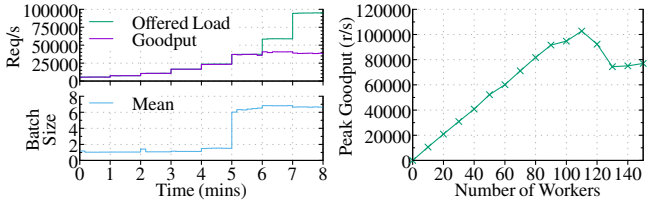


Fig. 11: (Left) With 40 emulated workers, goodput is approximately equal to offered load; peak goodput is achieved at appx. 40,000 r/s, when all workers are fully utilized. (Right) Peak goodput achieved with different numbers of emulated workers.

to exceed 10 ms in a few extremely rare cases. Clockwork consistently overpredicts more than it underpredicts, as it uses a rolling 99th percentile measurement to make its predictions. For LOAD actions, the 99th percentile of overpredictions and underpredictions is 431 μs and 348 μs, respectively.

Fig. 10 (bottom) plots the *completion time error*. Clockwork must accurately predict when a given action will complete, taking into account any previously submitted actions (*i.e.* queuing time). Individual prediction errors can compound, leading to increased completion time error. For INFER actions, the error compounds 4×, with a 99th percentile completion error of ≈1 ms. In extreme cases, Clockwork's completion error also grows to more than 10 ms. However, the completion error does not substantially exceed the action duration error, implying that for Clockwork, erroneous predictions of outliers are statistically independent.

6.6 Can Clockwork Scale?

Centralized scheduling presents a potential scalability bottleneck, though prior work has demonstrated that centralized schedulers can reach impressive scale [24, 57]. Our final experiment examines the scalability of Clockwork's controller.

To venture beyond the capacity of our testbed, we leverage a specially-developed *emulated worker* that implements Clockwork's action interface. The emulated worker behaves identically to a bona fide Clockwork worker, except the LOAD and INFER actions perform no meaningful work; instead, they wait for a period of time according to the pre-profiled model measurements before returning a response. The emulated worker is indistinguishable from a real worker from the vantage point of Clockwork's controller. To bypass the limited network capacity of our testbed, we modified our clients to send zero-length inputs (network is not a fundamental limitation; see §7 for discussion).

We measure the peak goodput achieved as we vary N, the number of emulated workers. We run multiple experiments, each with a different value of N, from 10 to 150 in increments of 10. We use the same models as described in §6.5, and a similar workload. Instead of replaying the trace at a fixed rate, we scale the trace and gradually offer more load in 60-second intervals. Fig. 11 (Left) illustrates one experiment run with N=40. Goodput follows the offered load almost perfectly up to about 40,000 r/s, at which point all workers are fully utilized and the goodput saturates.

Fig. 11 (Right) reports the peak goodput achieved with different numbers of workers. We report the median values across three experiment repetitions. The figure shows a linear increase in the peak goodput as the number of workers increases. Below N=110, goodput is limited by workers reaching full utilization. At N=110, we reach a maximum goodput of 103,387 r/s. At this point worker utilization stops being the limiting factor; instead, the bottleneck shifts to Clockwork's controller. Beyond N=110 peak goodput declines.

6.7 Summary

In comparison with prior model serving systems, Clockwork achieves superior goodput, serves considerably more models concurrently, and violates substantially fewer SLOs. Owing to a lack of performance variability, Clockwork can achieve much tighter latency SLOs without sacrificing tail latency. Clockwork's underlying assumptions about predictable executions bear out in reality: by consolidating choice, a predictable system that substantially curtails tail latency can be built.

Clockwork extends to a diverse range of workload conditions not supported by prior systems, including supporting thousands of models concurrently per GPU. Slow cold starts can run alongside high-throughput workloads without interference. Under all workload conditions, including cold starts and even under overload, Clockwork meets most SLOs without degrading service, and maintains close to maximal possible goodput. Finally, Clockwork isolates users of different models, enabling low-latency workloads to share the same system with background batch workloads.

7 Discussion

Why consolidate choice? Philosophically, the encapsulation, abstraction, and loose coupling of components are essential design practices while the building blocks and use cases of large systems are still in flux. Over time, the true use cases for the system settle and the entire system may in turn be replaced by a simpler, refined system that avoids the over-engineering and generality of its constituent parts—components that transpired to either be unnecessary in practice or to impede the common use case of the system. The squashing of layers through such specialization, effectively transforming systems into abstract units, can counteract the infamous bloat of modern software stacks. We designed Clockwork to be such an abstract unit for model serving systems.

Machine learning. Clockwork focuses on DNN inference, and excludes data preprocessing and postprocessing steps that are user-defined and CPU-bound. Safely and predictably executing these in Clockwork is a current research topic.

Individual DNN inferences are the atomic unit of work for Clockwork. Increasingly, modern ML applications are composed of pipelines or cascades of DNNs [34, 43, 62]. For these applications, performance predictability is strongly desired. We believe there are opportunities to leverage Clockwork's properties and perform more sophisticated pipeline scheduling that provides end-to-end guarantees. Similarly, performance predictability can influence system designs in other areas, such as large language embedding models [11] that may require dedicated or distributed accelerators. Expanding Clockwork into other ML paradigms, such as deep reinforcement learning and DNN training, raises philosophical questions about the nature and limits of predictability.

Inference accelerators. The Clockwork approach generalizes readily beyond GPUs to other inference-specific hardware accelerators [48], whose performance is arguably even more predictable. TPUs [41], for instance, are explicitly built around the idea of delegating control to software, while also eschewing general purpose processing engines with flexible control logic and generic memory hierarchies in favor of high-level operations and explicit memory hierarchies.

On the other extreme, inferences can also be executed in software on the CPU. While many models are heavily parallel in nature and execute orders of magnitude slower on CPUs, there are other models where execution on CPU is acceptable. One such example are recurrent neural networks (RNNs) which are fundamentally more sequential and often cannot effectively leverage the available parallelism on GPUs or other accelerators.

Limitations of predictability. Consolidating choice is only possible when you have control of, or guarantees about, the system's major bottleneck resources. For example, Clockwork assumes workers have exclusive control over their machine, and dedicated GPUs. Clockwork does not assume exclusive control over the network, but does assume that the network has mostly-predictable latency between the controller and workers. In a shared setting, preserving predictability becomes more challenging – though not impossible – and this is an active area of research due to a general need to co-locate latency critical datacenter services [42, 47].

Network. Clockwork does not explicitly consider the network in its scheduling decisions; the occasional network latency spikes of dozens of ms during our experiments had negligible impact on our results. Our prototype routes all inputs and outputs through the central controller which will become a bottleneck at scale. We were able to reach the limits of our testbed network with 12 workers and a sustained, single-model workload; to test beyond this we disabled inputs as described in §6.6. This limitation is not fundamental; Clockwork's controller only requires request *metadata* to schedule requests, and we are working to remove this limitation with a tier of load balancers.

Security. Security is important for all multi-user systems, since there are no container or hypervisor boundaries separating the workloads of different users. Clockwork does not explicitly address security; however, Clockwork does not execute arbitrary user code. Users must submit models in an abstract format that we then compile to binary code under the covers. Clockwork's threat model resembles shared storage or database

systems, where system correctness is the chief concern; we have not verified any safety properties of Clockwork.

Fault tolerance. While Clockwork is a distributed system, we do not address the challenges of tolerating failures when serving models at large scale.This will require implementing a fault-tolerant centralized scheduler; however, we note that Clockwork's predictable worker design will make pernicious phenomena like grey failure [27, 37] far easier to detect.

Other benefits of predictability. Concentrating choice makes it easier to implement other guarantees, such as SLOs related to burstiness or per-request cost. The Azure trace in our evaluation, for instance, contained regular, periodic spikes; exploiting advanced knowledge is an appealing future avenue for Clockwork. A further benefit of predictable system components is *performance clarity* [55]: performance bottlenecks and upcoming tasks in Clockwork are easy to reason about. Clockwork's controller also provides a central point for *explanation*, since the controller has complete visibility of the expected and actual request behavior.

8 Related Work

Model serving. We directly compared Clockwork to Clipper [16] and INFaaS [58] in §6.1; here we provide additional comments. Both Clipper and INFaaS are designed as wrappers around existing model execution frameworks: Clipper, in order to provide a unifying abstraction; INFaaS, in order to exploit heterogeneous execution strategies. Being agnostic to the underlying execution engine sacrifices predictability and control over model execution. Both systems treat latency SLOs as long-term, reactive targets; by contrast, Clockwork is explicitly designed to consolidate choice, and exploit predictability by making proactive decisions. Clipper and INFaaS propose several orthogonal concepts that are compatible with Clockwork. Clipper's model selection layer could be superimposed on Clockwork. INFaaS's model variant concept could be integrated into Clockwork; we found similar predictability properties held for DNNs executing on dedicated CPU cores.

Several other projects investigate model serving in virtualized cloud environments and on serverless platforms, where predictability is in the hands of the cloud provider [10, 44, 77]. Like INFaaS, these model throughput, latency, and accuracy together for optimal model selection, but, unlike Clockwork, they do not use the backend predictability and latency SLOs for making proactive scheduling decisions. In industry, TFS^2 [51] is a proprietary model hosting service at Google, about which public information is not available. Amazon SageMaker [59] and Google AI Platform [26] are public cloud DNN serving systems with a similar interface to Clockwork: upload your model, then make inference requests. Both use containers under the covers as an isolation mechanism, and users suffer the associated cold-start latency. Beyond these details, further design information is not publicly known.

Real-time systems. Performance predictability, especially temporal safety, is also an important concern for safety-critical real-time systems. However in general, real-time systems are designed for *periodic* or *sporadic* workloads [8] with known minimum inter-arrival times and worst-case execution times, or for scenarios where the set of all inference requests is known in advance [66]. *Soft-real-time* systems [12] consider weaker notions of timeliness similar to the latency SLOs considered in this paper, but mainly target periodic or sporadic workloads. Clockwork, in contrast, makes no *a priori* assumptions about its workloads. Prior real-time systems work has also proposed mechanisms to tame the unpredictability inside GPUs [6, 9, 20, 22, 54]. Elliott and Anderson [21], for example, proposed interrupt handling mechanisms to circumvent the proprietary GPU drivers that ignore scheduling priorities, while Yang *et al.* [74] suggested avoiding synchronization anomalies through more careful use of CUDA synchronization primitives. These mechanisms are designed to facilitate an *a priori schedulability analysis*— mathematically bounding the blocking delays due to contention. Such bounds are orthogonal to Clockwork, which does not require strict worst-case guarantees.

9 Conclusion

As DNN inferences become increasingly central to interactive applications, the requirements for fast response tighten, the volume of requests expands, and the number of models grows. Our model serving system, Clockwork, meets these challenges. Clockwork efficiently fulfills aggressive tail-latency SLOs while supporting thousands of DNN models with different workload characteristics concurrently on each GPU, and scaling out to additional worker machines for increased capacity. The system also successfully isolates models from performance interference caused by other models served on the same system. Our results derive from our design methodology of recursively ensuring all internal architecture components have predictable performance by concentrating all choices in the centralized controller. Notably, our approach required us to either circumvent canonical best-effort mechanisms or orchestrate them to become predictable, and illustrates how consolidating choice can be applied in practice to achieve predictable performance.

Acknowledgements

We thank our shepherd Junfeng Yang and the anonymous reviewers for their insightful feedback that helped improve our work. Our work was partially supported by NSF CAREER Grant #1553579.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Kudlur Manjunath, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan

Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[3] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. LASER: A Scalable Response Prediction Platform for Online Advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM)*, 2014.

[4] Saamer Akhshabi and Constantine Dovrolis. The Evolution of Layered Protocol Stacks leads to an Hourglass-Shaped Architecture. In *Proceedings of the 2011 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.

[5] Allied Market Research. Global machine learning chip market to garner $37.85 Billion by 2025, at 40.8% CAGR. `https://www.globenewswire.com/news-release/2020/02/18/1986370/0/en/Global-Machine-Learning-Chip-Market-to-Garner-37-85-Billion-by-2025-at-40-8-CAGR.html`, February 2020.

[6] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[7] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[8] Theodore P Baker and Sanjoy K Baruah. Schedulability analysis of multiprocessor sporadic task systems. *Handbook of Real-Time and Embedded Systems*, pages 3–31, 2007.

[9] Joshua Bakita, Nathan Otterness, James H Anderson, and F Donelson Smith. Scaling Up: The Validation of Empirically Derived Scheduling Rules on NVIDIA GPUs. In *14th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2018.

[10] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services. In *Proceedings of the 7th IEEE International Conference on Cloud Engineering (IC2E)*, 2019.

[11] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability Vs. Efficiency*. Springer Science & Business Media, 2005.

[13] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S Lam. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proceedings of the 26th International World Wide Web Conference (WWW)*, 2017.

[14] Wai Chee Yau. How Zendesk Serves TensorFlow Models in Production. `https://medium.com/zendesk-engineering/how-zendesk-serves-tensorflow-models-in-production-751ee22f0f4b`, February 2017.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[16] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[17] Brian Dalessandro, Daizhuo Chen, Troy Raeder, Claudia Perlich, Melinda Han Williams, and Foster Provost. Scalable Hands-Free Transfer Learning for Online Advertising. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2014.

[18] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[19] Christina Delimitrou and Christos Kozyrakis. Amdahl's Law for Tail Latency. *Communications of the ACM*, 61(8):65–72, 2018.

[20] Glenn A Elliott and James H Anderson. Real-world Constraints of GPUs in Real-Time Systems. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.

[21] Glenn A Elliott and James H Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[22] Glenn A Elliott and James H Anderson. An Optimal k-Exclusion Real-Time Locking Protocol Motivated by Multi-GPU Systems. *Real-Time Systems*, 49(2):140–170, 2013.

[23] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, 30(4):1–26, 2012.

[24] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[26] Google AI Platform. Retrieved May 2020 from `https://cloud.google.com/ai-platform/`, 2020.

[27] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage*, 14(3):1–26, 2018.

[28] Jian Guo, He He, Tong He, Leonard Lausen, Mu Li, Haibin Lin, Xingjian Shi, Chenguang Wang, Junyuan Xie, Sheng Zha, et al. GluonCV and GluonNLP: Deep Learning in Computer Vision and Natural Language Processing. *Journal of Machine Learning Research*, 21(23):1–7, 2020.

[29] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE 2016 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks. In *Proceedings of the 14th European Conference on Computer Vision (ECCV)*, 2016.

[32] Jeremy Hermann and Mike Del Balso. Meet Michelangelo: Uber's Machine Learning Platform. `https://eng.uber.com/michelangelo/`, September 2017.

[33] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for MobileNetV3. In *Proceedings of the IEEE 2019 Conference on Computer Vision (ICCV)*, 2019.

[34] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying Large Video Dataset with Low Latency and Low Cost. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[35] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-Excitation Networks. In *Proceedings of the IEEE 2018 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[36] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely Connected Convolutional Networks. In *Proceedings of the IEEE 2017 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[37] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[38] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[39] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving Deep Learning Models in a Serverless Platform. In *Proceedings of the 6th IEEE International Conference on Cloud Engineering (IC2E)*, 2018.

[40] Chris Jones, John Wilkes, Niall Murphy, and Cody Smith. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. `https://landing.google.com/sre/sre-book/chapters/service-level-objectives/`.

[41] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017.

[42] Kostis Kaffes, Dragos Sbirlea, Yiyan Lin, David Lo, and Christos Kozyrakis. Leveraging Application Classes to Save Power in Highly-Utilized Data Centers. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.

[43] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment*, 10(11), 2017.

[44] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, 2019.

[45] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In *Proceedings of the IEEE 2016 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[46] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[47] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[48] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[49] Neural Network Exchange Format (NNEF). Retrieved May 2020 from `https://www.khronos.org/nnef/`, 2020.

[50] NVIDIA TensorRT. Retrieved May 2020 from `https://developer.nvidia.com/tensorrt`, 2020.

[51] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. TensorFlow-Serving: Flexible, High-Performance ML Serving. *Workshop on ML Systems at NeurIPS 2017*, 2017.

[52] Open Neural Network Exchange Format: The new open ecosystem for interchangeable AI models. Retrieved May 2020 from `https://onnx.ai/`, 2020.

[53] The ONNX Model Zoo. Retrieved May 2020 from `https://github.com/onnx/models`, 2020.

[54] Nathan Otterness, Ming Yang, Tanya Amert, James Anderson, and F Donelson Smith. Inferring the scheduling policies of an embedded cuda gpu. In *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

[55] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[56] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[57] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. FastPass: A Centralized "Zero-Queue" Datacenter Network. In *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[58] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: A Model-less Inference Serving System. *arXiv preprint arXiv:1905.13348*, 2019.

[59] Deploying a Model on Amazon SageMaker Hosting Services. Retrieved May 2020 from `https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-hosting.html`, 2020.

[60] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *arXiv preprint arXiv:1907.10597*, 2019.

[61] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC '20)*, 2020.

[62] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU Cluster Engine for Accelerating DNN-based Video Analysis. In

Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), 2019.

[63] Julien Simon. Amazon Elastic Inference – GPU-Powered Deep Learning Inference Acceleration. `https://aws.amazon.com/blogs/aws/amazon-elastic-inference-gpu-powered-deep-learning-inference-acceleration/`, November 2018.

[64] Kacper Sokol and Peter A Flach. Glass-Box: Explaining AI Decisions With Counterfactual Statements Through Conversation With a Voice-enabled Virtual Assistant. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[65] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. *arXiv preprint arXiv:1906.02243*, 2019.

[66] Jinghao Sun, Jing Li, Zhishan Guo, An Zou, Xuan Zhang, Kunal Agrawal, and Sanjoy Baruah. Real-Time Scheduling upon a Host-Centric Acceleration Architecture with Data Offloading. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[67] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE 2015 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[68] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE 2016 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[69] Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Robert Burgess, Gregory Chockler, Haoyuan Li, and Yoav Tock. Dr. Multicast: *Rx* for Data Center Communication Scalability. In *Proceedings of the 5th European Conference on Computer systems (EuroSys)*, 2010.

[70] Limin Wang, Yuanjun Xiong, Zhe Wang, Yu Qiao, Dahua Lin, Xiaoou Tang, and Luc Van Gool. Temporal Segment Networks: Towards Good Pratices for Deep Action Recognition. In *Proceedings of the 14th European Conference on Computer Vision (ECCV)*, 2016.

[71] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine Learning at Facebook: Understanding Inference at the Edge. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[72] Bin Xiao, Haiping Wu, and Yichen Wei. Simple Baselines for Human Pose Estimation and Tracking. In *Proceedings of the 16th European Conference on Computer Vision (ECCV)*, 2018.

[73] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated Residual Transformations for Deep Neural Networks. In *Proceedings of the IEEE 2017 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[74] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H Anderson, and F Donelson Smith. Avoiding Pitfalls when using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[75] Fisher Yu, Dequan Wang, Evan Shelhamer, and Trevor Darrell. Deep Layer Aggregation. In *Proceedings of the IEEE 2018 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[76] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. *arXiv preprint arXiv:1605.07146*, 2016.

[77] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting Cloud Services for Cost-Effective, SLO-aware Machine Learning Inference Serving. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[78] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Zhi Zhang, Haibin Lin, Yue Sun, Tong He, Jonas Mueller, R Manmatha, et al. ResNeSt: Split-Attention Networks. *arXiv preprint arXiv:2004.08955*, 2020.

A Artifact Appendix

A.1 Abstract

The artifact consists of Clockwork's prototype source code, instructions for building from source, and directions for preparing the environment. The instructions for launching a Docker instance that has all dependencies pre-installed is provided as well. The artifact also contains scripts, descriptions, and instructions to run the experiments automatically or manually for reproducing the graphs and results presented in the paper.

A.2 Artifact check-list

- **Program:** dnn-model-serving, multi-tenant
- **Compilation:** cmake, g++
- **Binary:** worker, controller, client
- **Model:** distributed, multi-tenant
- **Data set:** azure-functions-trace-2019, poission-distribution
- **Run-time environment:** Linux, CUDA, network
- **Hardware:** NVIDIA, Tesla-V100
- **Execution:** automated, manual
- **Metrics:** throughput, latency, SLO-violation, tail-latency
- **Output:** telemetry-measurements, table, graph
- **Experiments:** throughput-latency, scalability, predictability, SLO, tail-latency
- **Required disk space:**
  Clockwork: 210MB
  Total including compiled models and dataset: 12GB
- **Expected experiment run time:** About 17 hours in total
- **Public link:**
  `https://gitlab.mpi-sws.org/cld/ml/clockwork`
- **Code licenses:**
  Clockwork: Apache License 2.0
  TVM: Apache License 2.0
  CUDA Common Library: Apache License 2.0
  Catch2: Boost Software License 1.0
- **Data licenses:**
  Azure Functions Trace 2019: CC-BY Attribution

A.3 Description

A.3.1 How to access

The artifact is publicly available at
`https://gitlab.mpi-sws.org/cld/ml/clockwork`

A.3.2 Hardware dependencies

To reproduce the exact experiment results, worker machines must have 768GB RAM or higher, 16 CPU cores or more, at least one 32GB Tesla v100 GPU and 10Gbps network. The large-scale experiment with Azure Functions (Fig. 9) requires 12 worker machines. Most other experiments require fewer worker machines; details on the number of machines for each experiment and environment customization guide is provided in each experiment's documentation.

A.3.3 Software dependencies

- **Clockwork:**
  Ubuntu 18.04 or later, CUDA v9.0+, libtbb-dev, libasio-dev, libconfig++-dev, libboost-all-dev, g++-8, make, cmake, automake, autoconf, libtool, curl, unzip, clang, llvm, and protobuf.

  A Dockerfile is provided to facilitate the build process.

- **Data analysis and plotting scripts:**
  Python 3.x and the numpy, pandas, matplotlib, and seaborn libraries.

A.3.4 Data sets

- Publicly released Azure Functions 2019 trace [61]
  `https://gitlab.mpi-sws.org/cld/trace-datasets/azure-functions`

A.3.5 Models

The DNN models pre-compiled for NVIDIA Volta V100 GPUs are accessible at
`https://gitlab.mpi-sws.org/cld/ml/clockwork-modelzoo-volta`

A.4 Installation

- Installation pre-requisites:
  `https://gitlab.mpi-sws.org/cld/ml/clockwork/-/blob/master/docs/prerequisites.md`

- Building Clockwork:
  `https://gitlab.mpi-sws.org/cld/ml/clockwork/-/blob/master/docs/building.md`

- Setting-up the environment:
  `https://gitlab.mpi-sws.org/cld/ml/clockwork/-/blob/master/docs/environment.md`

- Clockwork configuration:
  `https://gitlab.mpi-sws.org/cld/ml/clockwork/-/blob/master/docs/configuration.md`

A.5 Experiment workflow

Experiments can be run using the scripts provided in the repository. We have also provided instructions to run the experiments manually. To get started with Clockwork, we recommend getting the system running manually, in order to understand the pieces involved, and to ensure the system has been configured appropriately for your machines. Afterwards, you might choose to run the experiments using the provided scripts or manually. The experiments repository is available at
`https://gitlab.mpi-sws.org/cld/ml/clockwork-results`

Experiment	Related figure	Execution time (hr)	Documentation and scripts
How Does Clockwork Compare?	Fig. 5	3	https://gitlab.mpi-sws.org/cld/ml/clockwork-results/-/tree/master/sec61_fig5
Can Clockwork Serve Thousands?	Fig. 6	1.5	https://gitlab.mpi-sws.org/cld/ml/clockwork-results/-/tree/master/sec62_fig6
How Low Can Clockwork Go?	Fig. 7	1	https://gitlab.mpi-sws.org/cld/ml/clockwork-results/-/tree/master/sec63_fig7
Can Clockwork Isolate Performance?	Fig. 8	1	https://gitlab.mpi-sws.org/cld/ml/clockwork-results/-/tree/master/sec64_fig8
Are Realistic Workloads Predictable?	Fig. 10	8	https://gitlab.mpi-sws.org/cld/ml/clockwork-results/-/tree/master/sec65_fig9_fig10
Can Clockwork Scale?	Fig. 11	2	https://gitlab.mpi-sws.org/cld/ml/clockwork-results/-/tree/master/sec66_fig11

Table 3: The experiments reproducing the presented results in this paper, their related figures, execution time, and links to the extensive documentation and scripts for each experiment.

A.6 Evaluation and expected results

The experiments repository is structured based on §6. We have provided the experiment titles, their related figures on the paper, execution time of each experiment, and the links to directories containing the respective descriptions, scripts and instructions in Table 3.

A.7 Experiment customization

The directions for running each experiment manually is provided in each experiment's documentation. Instructions for customizing the experiment environment is provided at `https://gitlab.mpi-sws.org/cld/ml/clockwork/-/blob/master/docs/customizing.md`

A.8 AE Methodology

Submission, reviewing and badging methodology: `https://www.usenix.org/conference/osdi20/call-for-artifacts`



A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters

Yimin Jiang*†, Yibo Zhu†, Chang Lan‡, Bairen Yi†, Yong Cui*, Chuanxiong Guo†
*Tsinghua University, †ByteDance, ‡Google

Abstract

Data center clusters that run DNN training jobs are inherently heterogeneous. They have GPUs and CPUs for computation and network bandwidth for distributed training. However, existing distributed DNN training architectures, all-reduce and Parameter Server (PS), cannot fully utilize such heterogeneous resources. In this paper, we present a new distributed DNN training architecture called BytePS. BytePS can leverage spare CPU and bandwidth resources in the cluster to accelerate distributed DNN training tasks running on GPUs. It provides a communication framework that is both *proved optimal* and unified – existing all-reduce and PS become two special cases of BytePS. To achieve the proved optimality in practice, BytePS further splits the functionalities of a parameter optimizer. It introduces a *Summation Service* abstraction for aggregating gradients, which is common for all the optimizers. Summation Service can be accelerated by AVX instructions and can be efficiently run on CPUs, while DNN model-related optimizer algorithms are run on GPUs for computation acceleration. BytePS can accelerate DNN training for major frameworks including TensorFlow, PyTorch and MXNet. For representative DNN training jobs with up to 256 GPUs, BytePS outperforms the state-of-the-art open source all-reduce and PS by up to 84% and 245%, respectively.

1 Introduction

In recent years, research on Deep Neural Networks (DNNs) has experienced a renaissance. DNNs have brought breakthroughs to computer vision [32, 43], speech recognition and synthesis [33, 69], natural language processing (NLP) [26], and many other areas. Training these DNN models usually requires a huge amount of arithmetic computation resources. Consequently, GPUs are preferred. To run many such tasks and achieve high resource utilization, large GPU clusters with thousands or more GPUs are introduced [29, 35, 52, 71].

Such GPU clusters have not only GPUs, but also CPUs and high speed networks. GPU machines typically also have high-end CPUs [2, 11]. There may also be CPU-only machines used for training data pre-processing and generation, *e.g.,* in reinforcement learning. These GPU/CPU machines are connected by high-speed Ethernet or Infiniband network to facilitate distributed training. Based on our experience in operating production GPU clusters (§3.1) and recent literature from others [35], GPUs are usually better utilized while there are often spare CPU and bandwidth resources.

There are two major families of distributed training architectures, all-reduce [54] and Parameter Server (PS) [44]. They are both based on data parallelism (§2). In a task that uses all-reduce, only GPU machines are involved. In an iteration, GPUs compute the gradients of the model parameters independently, and then aggregate gradients using the all-reduce primitive. In PS tasks, both GPU machines and CPU machines can be used. Different from all-reduce, the gradients are sent to PS, which typically runs on CPU machines and aggregates the received gradients. PS then runs certain DNN training optimizer, *e.g.,* SGD [76] or Adam [42] and sends back the updated model. For both all-reduce and PS, the above happens in every iteration, until the training finishes.

All-reduce and PS are quite different, in both theory and practice. Given a set of GPU machines *without* additional CPU machines, all-reduce is proved to be bandwidth optimal [54]. However, *with* additional CPU and bandwidth resources, the optimality of all-reduce no longer holds – we find that, *in theory*, PS can offer even better performance by utilizing additional CPU machines to aid the GPU machines (§2). It seems to be a good opportunity to accelerate DNN training because GPU clusters indeed have spare CPU and bandwidth resources (§3.1). Unfortunately, *in practice,* all the existing PS have inferior performance for multiple design reasons, as we shall see soon in this paper. It is therefore not a surprise to see that distributed DNN training speed records are dominated by all-reduce [27, 49, 73].

We are thus motivated to design *BytePS* [1], an architecture that is communication-optimal, both in theory and in practice. Fundamentally, both all-reduce and PS are theoretically optimal only in very specific GPU/CPU setups, while are not

[1]The name BytePS was chosen in the early stage of this project [4]. However, it is conceptually different from the conventional PS architecture.

the optimal for more generic settings, *e.g.,* there are some finite additional CPU resources. By carefully allocating traffic loads, BytePS unifies the cases where PS or all-reduce is theoretically optimal, and generalizes the optimality to any given number of GPU/CPU machines with different PCIe/NVLink configurations, with analytical proofs.

On top of that, BytePS pushes its real-world performance close to the theoretical limit, by removing bottlenecks in existing PS designs. With fast high-speed networks, we found that CPUs are not fast enough for the full fledged DNN optimizers. We introduce a new abstraction, *Summation Service*, to address this issue. We split an optimizer into gradient aggregation and parameter update. We keep gradient aggregation in Summation Service running on CPUs and move parameter update, which is more computation intensive, to GPUs. In addition, in implementation, we incorporated the idea of pipelining and priority-scheduling from prior work [34, 55] and resolved multiple RDMA-related performance issues.

As a drop-in replacement for all-reduce and PS, BytePS aims to accelerate distributed training without changing the DNN algorithm or its accuracy at all. Prior work on top of all-reduce and PS, like tensor compression [21, 45], can directly apply to BytePS. Our BytePS implementation supports popular DNN training frameworks including TensorFlow [20], PyTorch [53], and MXNet [22] with Horovod-like [60] API and native APIs.

This paper makes the following contributions:

- We design a new distributed DNN training architecture, BytePS, for heterogeneous GPU/CPU clusters. With spare CPU cores and network bandwidth in the cluster, BytePS can achieve communication optimality [2] for DNN training acceleration. BytePS provides a unified framework which includes both all-reduce and PS as two special cases.
- We further optimize the intra-machine communication. We explain the diverse and complicated topology in GPU machines and present the optimal strategy and principles.
- We propose Summation Service, which accelerates DNN optimizers by keeping gradient summation running in CPUs, and moving parameter update, which is the more computation intensive, to GPUs. This removes the CPU bottleneck in the original PS design.

As a major online service provider, we have deployed BytePS internally and used it extensively for DNN training. We evaluate BytePS using six DNN models and three training frameworks in production data centers. The results show that with 256 GPUs, BytePS consistently outperform existing all-reduce and PS solutions by up to 84% and 245%, respectively. We also released an open source version [4], which attracted interests from thousands in the open source community, several top-tier companies and multiple research groups.

[2]The optimality means to achieve minimized communication time for data-parallel distributed DNN training, given a fixed number of GPUs.

2 Background

2.1 Distributed DNN Training

A DNN model consists of many *parameters*. DNN training involves three major steps: (1) *forward propagation (FP)*, which takes in a *batch* of training data, propagates it through the DNN model, and calculates the *loss function*; (2) *backward propagation (BP)*, which uses the loss value to compute the *gradients* of each parameter; (3) *parameter update*, which uses the aggregated gradients to update the parameters with a certain optimizer (e.g., SGD [76], Adam [42], etc.). Training a DNN refines the model parameters with the above three steps iteratively, until the loss function reaches its minimal.

On top of it, users can optionally run distributed training. The most popular distributed DNN training approach is *data parallelism*, which partitions the dataset to multiple distributed computing devices (typically GPUs) while each GPU holds the complete DNN model. Since the data input to each GPU is different, the gradients generated by BP will also be different. Thus data parallelism demands all GPUs to synchronize during each training iteration.

In large enterprises or in public clouds, users often run these DNN training tasks in shared GPU clusters. Such clusters are built with hundreds to thousands of GPU machines connected by high-speed RDMA networks [35, 52]. Those GPU machines typically have multiple GPUs, tens of CPU cores, hundreds of GB of DRAM, and one to several 100Gb/s NICs. These clusters run many training jobs simultaneously, with many jobs using GPUs intensively while not using CPUs heavily. A public dataset on a DNN cluster [35] indicates that 50% of hosts have CPU utilization lower than 30%.

For distributed training, there are two families of data parallelism approaches, *i.e.*, all-reduce and Parameter Server (PS). In what follows, we introduce all-reduce and PS and analyze their communication overheads. We assume that we have n GPU machines for a data-parallel training job. The DNN model size is M bytes. The network bandwidth is B.

2.2 All-reduce

Originated from the HPC community, *all-reduce* aggregates every GPU's gradients in a collective manner before GPUs update their own parameters locally. In all-reduce, no additional CPU machine is involved. *Ring* is the most popular all-reduce algorithm. All-reduce has been optimized for many years, and most state-of-the-art training speed records are achieved using all-reduce, including classical CNN-based ImageNet tasks [27, 36, 49, 73], RNN-based language modeling tasks [56], and the pre-training of BERT [26, 74].

Fig. 1 shows an example of ring-based all-reduce for three nodes. We can dissect an all-reduce operation into a *reduce-scatter* and an *all-gather*. Reduce-scatter (Fig. 1(a)) partitions the whole M bytes into n parts, and use n rings with different starting and ending point to reduce the n parts, respectively. Each node will send $(n-1)M/n$ traffic, because each node

acts as the last node for just 1 ring and thus sends 0, while for each of the other $n-1$ rings, it must send M/n bytes.

Next, all-gather requires each node to broadcast its reduced part to all other $(n-1)$ nodes using a ring. In the end, all nodes have identical data that have been all-reduced (Fig. 1(c)). Similar to reduce-scatter, each node also sends $(n-1)M/n$ egress traffic during this operation.

Adding the two steps together, in an all-reduce operation, each node sends (and receives) $2(n-1)M/n$ traffic to (and from) the network. With B network bandwidth, the time required is $2(n-1)M/nB$, which is proved to be the optimal in topologies with uniformed link bandwidth [54], assuming *no additional resources*.

In hierarchical topologies with non-uniformed link bandwidth, the optimal hierarchical strategy would require at least $2(n'-1)M/n'B'$ communication time, where B' is the slowest link bandwidth and n' is the number of nodes with the slowest links. In distributed DNN training, n' is usually the number of GPU machines and B' is usually the network bandwidth per machine. For simplicity and without impacting our analysis, below we assume each machine has just one GPU and is connected by the same network bandwidth, *i.e.,* $n=n', B=B'$.

All-reduce has no way to utilize *additional* non-worker nodes, since it was designed for homogeneous setup. Next, we will show that the $2(n-1)M/nB$ communication time is no longer optimal with additional CPU machines.

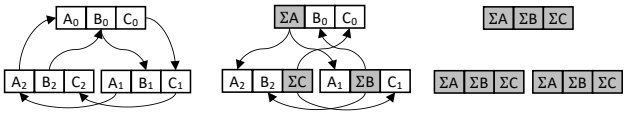
2.3 Parameter Server (PS)

The PS architecture [44] contains two roles: *workers* and *PS*. Workers usually run on GPU machines, perform FP and BP, and *push* the gradients to PS. PS aggregates the gradients from different workers and update the parameters. Finally, workers *pull* the latest parameters from PS and start the next iteration. According to our experience in industry, the PS processes usually run on CPUs because of cost-effectiveness. Since GPUs (and GPU memory) are much more expensive than CPUs,[3] we want GPUs to focus on the most computation-intensive tasks instead of storing the model parameters.

There are two placement strategies for PS. One is *non-colocated mode* (Fig. 2(a)), in which PS processes are deployed on dedicated CPU machines, separate from the GPU machines. Suppose that we have k CPU machines,[4] the DNN model will be partitioned into k parts and stored on the k machines, respectively. In every iteration, each GPU worker must send M bytes gradients and receives M bytes parameters back. Each CPU machine must receive in total nM/k gradients from the GPU workers and send back nM/k parameters.

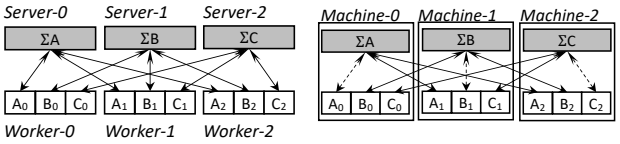
[3] AWS price sheet [18] shows that p3.16xlarge (8 NVIDIA V100 GPUs and 64 CPU cores) costs nearly $25 per hour. However, r4.16xlarge, which is the same as p3.16xlarge minus GPUs, costs only $4.2 per hour.

[4] In this paper, for simplicity, we assume that a CPU machine has the same network bandwidth as a GPU machine. If not, all analysis and design will remain valid as long as the number of CPU machines scales accordingly. For example, use 4× CPU machines if their bandwidth is 25% of GPU machines.



(a) Reduce-scatter (b) All-gather (c) Result

Figure 1: The communication workflow of all-reduce.



(a) Non-colocated mode (b) Colocated mode

Figure 2: The communication pattern of PS. A solid arrow line indicates the network traffic. A dashed arrow line represents the loop-back (local) traffic.

Assuming $k=n$, PS would theoretically be faster than all-reduce, as summarized in Table 1. In fact, PS is communication optimal in such setting, since M is the absolute lower bound each GPU machine has to send and receive. However, with fewer CPU machines (smaller k), the communication time nM/kB on CPU machines would increase and, if $k \le n/2$, become slower than all-reduce. The network bandwidth of GPU machines would become under-utilized because the CPU machines would be the communication bottleneck.

The other strategy is *colocated mode* (Fig. 2(b)), which does not use any CPU machines. Instead, it starts a PS process on every GPU worker and reuses its spare CPU resources. The PS and GPU worker on the same machine will communicate through loopback traffic. In this case, it is easy to calculate that communication time is the same as all-reduce (Table 1).

All-reduce vs. PS. They have different communication patterns. PS uses a bipartite graph. Non-colocated PS can leverage additional CPU and bandwidth resources to aid GPU machines, while may under-utilize the resources of GPU machines. Colocated PS and all-reduce utilize the GPU worker resources better, while cannot use additional CPU machines.

Another difference is that PS supports *asynchronous* training, which allows GPU workers to run at different speed and mitigates the impact of stragglers, while all-reduce does not support it. However, asynchronous training is less popular because it can slow down model convergence. We will mainly focus on synchronous training in this paper while briefly address asynchronous training in §5.

3 Motivation and BytePS Architecture

3.1 Motivation

Before the deployment of BytePS in our internal GPU clusters, our users mostly used all-reduce as the distributed training architecture due to its higher performance than existing PS designs. The remaining users choose PS for tasks where asynchronous training is acceptable or preferable. With multiple years of experience and efforts on accelerating DNN tasks and improving resource utilization, we have the following observation.

Table 1: The theoretical communication time required by each training iteration. n is the number of GPU machines. k is the number of additional CPU machines. M is the model size. B is the network bandwidth. We will revisit the *Optimal?* row in §4.1.

	All-reduce	Non-Colocated PS	Colocated PS
Time	$\frac{2(n-1)M}{nB}$	$\max(\frac{M}{B}, \frac{nM}{kB})$	$\frac{2(n-1)M}{nB}$
Optimal?	Only if $k=0$	Only if $k=n$	Only if $k=0$

Opportunity: there are spare CPUs and bandwidth in production GPU clusters. Large-scale GPU clusters simultaneously run numerous jobs, many of which do not heavily use CPUs or network bandwidth. Fig. 3 shows a 3-month trace collected from one of our GPU clusters that have thousands of GPUs. The GPUs have been highly utilized in that period (approaching 96% allocation ratio in peak times). We find that, 55%-80% GPU machines have been assigned as GPU workers for at least one *distributed* training task. This leaves the network bandwidth of 20%-45% GPU machines unused because they are running *non-distributed* jobs.[5] The cluster-wide average CPU utilization is only around 20%-35%. This aligns with the findings in prior work from Microsoft [35].

This observation, combined with the all-reduce vs. non-colocated PS analysis in §2.1, inspires us – if we can better utilize these spare CPUs and bandwidth, it is possible to accelerate distributed training jobs running on given GPUs.

Existing all-reduce and PS architectures are insufficient. Unfortunately, the analysis in §2.1 also shows that all-reduce and PS have a common issue: they do not utilize additional CPU and bandwidth resources well. All-reduce and colocated PS only use resources on GPU workers, and non-colocated PS may not fully utilize the CPU cores and NIC bandwidth on GPU workers. The former is communication optimal only when $k=0$, while the latter is optimal only when $k=n$. When the number of CPU machine k is $0<k<n$, neither would be optimal. We defer further analysis to §4.1. Here, we use an experiment to show the end-to-end performance of existing all-reduce and PS.

Fig. 4 shows the training speed of VGG-16 [63] using 32 V100 GPUs (4 GPU machines), with 100GbE RDMA network. The batch size is 32 images for each GPU. We run the latest MXNet native PS RDMA implementation [1] and (one of) the most popular all-reduce library NCCL-2.5.7 [13]. We also tested TensorFlow's native PS, and got similar results. We vary the number of additional CPU machines for each setup. All-reduce plot is flat because additional CPU machines are of no use, while PS has the worst performance even with additional CPU machines. Both of them are far from optimal. Even with ByteScheduler [55], which is a state-of-the-art technique that can improve the communication performance, both all-reduce and PS are still far from the linear scaling, *i.e.,* 32× of single-GPU training speed. This is because ByteScheduler works on top of PS or all-reduce, and thus has the same limitations. BytePS outperforms all of above at any given number of CPU machines (more in §7).

[5] Our machines have dedicated but slower NIC for data I/O. This is a common practice in industry [52]. In addition, data I/O traffic is usually much smaller than the distributed training traffic between GPU machines.

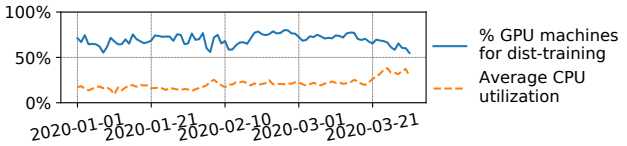


Figure 3: Daily statistics of our internal DNN training clusters from 2020-01-01 to 2020-03-31.

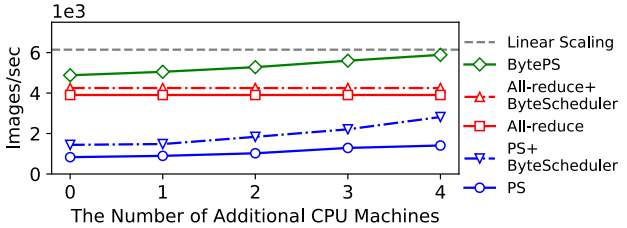


Figure 4: VGG-16 training performance of different architectures. We use 4 GPU machines with 32 GPUs in total. *Linear Scaling* represents the maximal performance (in theory) of using 32 GPUs.

Our solution: BytePS. It is a unified architecture for distributed DNN training that can leverage spare CPU and bandwidth resources. It achieves the following goals.

First, BytePS is always communication optimal with any additional CPU and bandwidth resources, *i.e.,* $0 \le k \le n$, allocated by the cluster scheduler. In practice, the volume of spare resources can be dynamic (Fig. 3), so BytePS must adapt well. In addition, the hardware setup of GPU machines can be diverse, especially the internal PCIe or NVLink topology. BytePS is also *proved* optimal in intra-machine communication. All-reduce and PS, when they are communication optimal, are two special cases of BytePS (§4).

Second, BytePS can achieve communication time very close to the theoretical optimal. This is important, as shown in the existing PS case – PS performance is far from its theoretical limit. We found that original PS designs have several implementation bottlenecks (which we will discuss in §6). But even after all the bottlenecks are removed, PS performance is still inferior to optimal. This leads to BytePS's second design contribution: *Summation Service*. We find that running the full optimizers on CPU can be a bottleneck. We divide the computation of optimizers and only put summation on CPUs. We will elaborate the rationale of this design in §5.

All the BytePS designs are generic to DNN training. BytePS can therefore accelerate various DNN training frameworks including TensorFlow, PyTorch, and MXNet. We start from presenting BytePS's architecture.

3.2 Architecture Overview

Fig. 5 shows the architecture of BytePS. BytePS has two main modules – *Communication Service (CS)* and *Summation Service (SS)*. In BytePS, we aim to leverage any CPU resources, whether on GPU machines or CPU machines, to

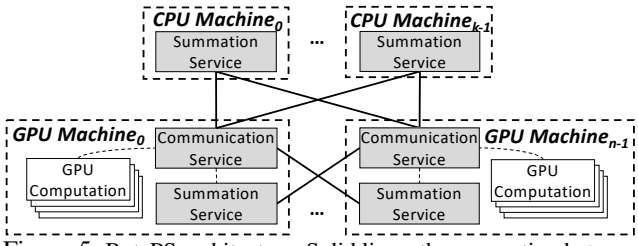


Figure 5: BytePS architecture. Solid lines: the connection between CPU machines and GPU machines. Dashed lines: the data flow inside GPU machines.

achieve the best communication efficiency. This is achieved by SS, which runs on the CPU of *every* machine, including the CPU machines and GPU machines. The CPU machines may not necessarily be actual CPU-only machines. For example, our in-house cluster scheduler can allocate CPUs on the GPU machines that run non-distributed jobs and have spare CPU cores and network bandwidth. This improves the overall cluster resource utilization.

Another important property of SS is that it is much simpler than common PS server processes, which run full fledged DNN algorithm optimizers. In contrast, SS is only responsible for receiving tensors that are sent by CS, summing up the tensors and sending them back to CS.

The other module, CS, is responsible for internally synchronizing the tensors among multiple (if there are) local GPUs and externally communicating with SS. Every training iteration, each CS must send in total M bytes (the DNN model size) to and receive M bytes from SS. In synchronous distributed training, the tensors are model gradients.

CS contains several design points of BytePS. First, it decides the traffic volume to each SS (both internal and external). The load assignment strategy is based on our analysis of the optimal communication strategy (§4.1). Second, it chooses the best local tensor aggregation strategy depending on different internal GPU and NIC topology (§4.2) of the GPU machines. Finally, both CS and SS should be optimized for RDMA in modern high-speed data centers (§6.2).

This architecture enables BytePS to flexibly utilize any number of additional CPU resources and network bandwidth. When the number of CPU machines is 0, *i.e.,* $k = 0$, the communication will fallback to only using SSs on GPU machines. When the number of CPU machines is the same as GPU machines, BytePS is as communication optimal as non-colocated PS. In other cases, BytePS can leverage SSs on all machines together. In fact, our analytical results will reveal the optimal communication strategy with any number of CPU machines, while PS and all-reduce are just two specific points in the whole problem space.

4 BytePS Communication Design

4.1 Inter-machine Communication

In BytePS, all networking communication is between CS and SS. To prevent a bottleneck node from slowing down the whole system, we must balance the communication time of all machines. In what follows, we assume the network has full bisection bandwidth, which is a common practice in deep learning clusters [52]. We also assume that the full bisection bandwidth can be fully utilized due to the newly introduced RDMA congestion control algorithms, e.g., DCQCN [75].

On each CPU machine, the summation workload of its SS determines the network traffic. For example, if a SS is responsible for summing up $x\%$ of the DNN model, the CPU machine would send and receive $x\% \times M$ bytes traffic to *every* GPU machine during each training iteration. However, the network traffic of a GPU machine is determined by the combination of CS and SS running on it. Due to this difference, BytePS classifies SS into SS_{CPU} and SS_{GPU} based on whether they run on CPU machines or GPU machines.

To minimize the communication time, BytePS assigns $M_{SS_{CPU}}$ bytes summation workload to each SS_{CPU}. $M_{SS_{CPU}}$ is given in Eq. 1, where $k \geq 1$ is the number of CPU machines and $n \geq 2$ is the number of GPU machines, and $k \leq n$. Outside these constraints, the communication time of BytePS falls back to trivial solutions like PS (when $k > n$) and all-reduce (when $k = 0$), as §4.1.1 shows.

$$M_{SS_{CPU}} = \frac{2(n-1)}{n^2+kn-2k}M \tag{1}$$

Similarly, BytePS assigns $M_{SS_{GPU}}$ bytes to each SS_{GPU}.

$$M_{SS_{GPU}} = \frac{n-k}{n^2+kn-2k}M \tag{2}$$

Eq. 1 and Eq. 2 show the workload assignment strategy that is optimal for minimizing the communication time. The analysis is in §4.1.1. In practice, the DNN model consists of tensors with variable sizes and may not allow us to perfectly assign workloads. BytePS uses an approximation method. It partitions the tensors into small parts no larger than 4MB.[6] Then, all CSs consistently index each part and hash the indices into the range of $[0, n^2+kn-2k)$. CSs will send and receive tensors to SSs based on the hash value and approximate the probabilities according to Eq. 1 and Eq. 2. Consistent indexing and hashing guarantee that the same part from all GPUs will be sent to and processed by the same SS.

4.1.1 Communication Efficiency Analysis

Next, we present the communication time analysis of BytePS. To simplify the analysis, we assume that the model size M is much larger than the partition size (4MB in our case). Partitioning enables BytePS not only to better balance the summation workloads, but also to well utilize the bidirectional network bandwidth by pipelining sending and receiving, as shown in [34, 55]. So, we further assume that sending and receiving the whole M bytes can fully overlap with negligible overhead. We have the following result.

[6] While we find that 4MB partition size works reasonably well in our environment, BytePS allow users to tune the partition size value.

Theorem 1. *The SS workload assignment given by Eq. 1 and Eq. 2 is optimal for minimizing communication time.*

Proof. We first consider the network traffic of a GPU machine. It runs a CS module and an SS module. CS should send and receive M bytes in total. However, when it communicates with the SS on the same GPU machine, the traffic does not go over the network. So, a CS module will send and receive $M - M_{SS_{GPU}}$ bytes. An SS module on a GPU machine must receive and send $M_{SS_{GPU}}$ from other $n-1$ GPU machines, *i.e.,* $(n-1)M_{SS_{GPU}}$ in total. Adding them together, a GPU machine with network bandwidth B requires communication time t_g:

$$t_g = \frac{M + (n-2)M_{SS_{GPU}}}{B} \tag{3}$$

Similarly, if $k > 0$, we can get that a CPU machine with network bandwidth B requires communication time t_c:

$$t_c = M_{SS_{CPU}}/B \tag{4}$$

In addition, the sum of all the SS workload should be equal to the total model size.

$$M = kM_{SS_{CPU}} + nM_{SS_{GPU}} \tag{5}$$

From Eq. 5, it is clear that the larger $M_{SS_{CPU}}$ is, the smaller $M_{SS_{GPU}}$ is. Consequently, when $n \geq 2$, the larger t_c is, the smaller t_g is (or t_g is unchanged if $n = 2$). In addition, we know that the final communication time is $\max(t_c, t_g)$.

To minimize the communication time, t_c and t_g need to be equal. If they are not equal, say $t_c > t_g$, it means the communication time can be further reduced by decreasing $M_{SS_{CPU}}$ and thus bring down t_c.

We let $t_c = t_g$ and combine Eq. 3, Eq. 4, and Eq. 5. Solving the equations with $M_{SS_{GPU}}$ and $M_{SS_{CPU}}$ as variables, we can get the optimal values as given by Eq. 1 and Eq. 2. □

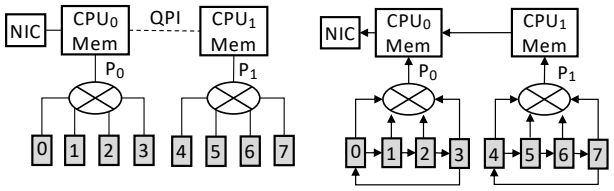
Based on Theorem 1, combine Eq. 3 and Eq. 2, we have the optimal communication time, which is used in Fig. 12.

$$t_{opt} = \frac{2n(n-1)M}{(n^2 + kn - 2k)B} \tag{6}$$

From Eq. 2, we can see that when the numbers of CPU machines and GPU machines are the same, $M_{SS_{GPU}} = 0$, which means that we do not need any SS_{GPU}. This is because the CPU machines already provide enough aggregate bandwidth. BytePS falls back to non-colcated PS. Similarly, when the number of CPU machines is 0, BytePS falls back to all-reduce and colocated PS.

Of course, the more interesting case is the general case when $0 < k < n$. We use the communication time of the plain all-reduce and non-colocated PS as the two baselines. We define the acceleration ratio γ_a as the communication time of the plain all-reduce divided by that of the general case. Similary, γ_p is defined as the acceleration ratio compared to the non-colocated PS case. We have

$$\gamma_a = \frac{n^2 + kn - 2k}{n^2}, \gamma_p = \frac{n^2 + kn - 2k}{2k(n-1)} \tag{7}$$



(a) PCIe-only topology (b) Outgoing data flow

Figure 6: PCIe-only machine topology and BytePS data flow. Gray boxes are GPUs. Only the outgoing direction (from GPUs to network) is shown in the data flow figure. Incoming is the opposite.

When $k = n$ and $n \to \infty$, $\gamma_a = 2$. When k is small, γ_p can be quite big, as the communication bandwidth is severely bottlenecked by the CPU machines in non-colocated PS. For example, when $n = 32$ and $k = 16$, we have $\gamma_a = 1.46$ and $\gamma_p = 1.52$, respectively. It means that BytePS can theoretically outperform all-reduce and PS by 46% and 52%, respectively.

We note that adding more CPU machines beyond $k = n$ does not help, since the communication bottleneck will become the NIC bandwidth of the GPU machines.

4.2 Intra-machine Communication

In §4.1, we design the optimal inter-machine communication strategy. In practice, we find that intra-machine communication is equally important. There are often multiple GPUs in a machine. CS must aggregate/broadcast the tensors before/after communicating with SS. This can create congestion on the PCIe links and prevent NIC from fully utilizing its bandwidth B. Moreover, the GPU machine's internal topology can be diverse in data centers. Below, we share the two most common machine setups in our environment and our corresponding solution. We present several principles that can apply to other machine setups in §4.2.3.

4.2.1 PCIe-only Topology

Fig. 6(a) shows a setup in our production environment. A GPU machine has two NUMA CPUs connected via QPI. The eight GPUs are split into two groups and connected to two PCIe switches, respectively. The NIC is 100Gbps and connected to the PCIe of one of the CPUs. All PCIe links in figure are 3.0 x16 (128Gbps theoretical bandwidth). The CPU memory and QPI has $> 300Gbps$ bandwidth, which are less likely the communication bottleneck. We call this *PCIe-only topology*. For this machine model, we measure that the throughput of GPU-to-GPU memory copy is ≈105Gbps within the same PCIe switch. The throughput of GPU-to-GPU memory copy across PCIe switches, however, is only ≈80Gbps.

Unfortunately, many existing training frameworks ignore such details of internal topology. For example, TensorFlow PS, MXNet PS and even the "hierarchical all-reduce" mode of Horovod use a straightforward reduce or reduce-scatter across *all* GPUs on the same machine. This would lead to cross-PCIe switch memory copy, which is unfortunately slower.

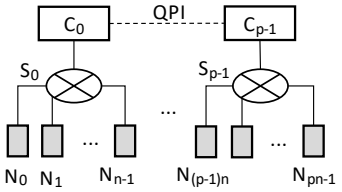


Figure 7: Notations of the PCIe-only topology.

In contrast, BytePS lets GPUs under the same PCIe switch sum the tensors first, then copy to CPU and let CPU do the global summation, and finally broadcast back the global sum. We call it *CPU-assisted aggregation.* Specifically, it consists of the following steps.

1. **Reduce-Scatter**: Suppose each PCIe switch has l GPUs. These l GPUs perform a reduce-scatter which incurs $(l-1)M/l$ traffic only inside the PCIe switch. When it finishes, each GPU should hold M/l aggregated data.
2. **GPU-CPU Copy**: Each GPU copies its M/l data to CPU memory, which incurs M/l traffic along the route. Every PCIe switch would generate M aggregated data.
3. **CPU-Reduce**: CPU reduces the data from all PCIe switches and generates the aggregated data across all GPUs. This reduction does not incur any PCIe traffic.
4. **Networking**: CS sends the data to SS and receives globally aggregated data from SS.
5. **CPU-GPU Copy**: Each GPU copies its M/l partition from CPU memory back to itself. This incurs M/l traffic from the CPU to each GPU.
6. **All-Gather**: Each GPU performs an all-gather operation with those that are under the same PCIe switch. This incurs $(l-1)M/l$ traffic inside the switch.

Fig. 6(b) shows the traffic of step 1 to 3. Step 4 to 6 use the same links but the opposite direction. With CPU-assisted aggregation, the PCIe switch to CPU link would carry only M traffic in each direction, much lower than doing collective operation directly on eight GPUs ($7M/4$ traffic). Meanwhile, the traffic on each PCIe switch to GPU link would be $(2l-1)M/l$. Let $l=4$ (each PCIe has four GPUs), this is $7M/4$, remaining the same as the existing approach. Fundamentally, BytePS leverages the spare CPUs on the GPU machine to avoid the slow GPU-to-GPU cross-PCIe switch memory copy.

Optimality Analysis. We now analyze the communication optimality of the above strategy. Fig. 7 shows a more generic PCIe-only topology with variable number of GPUs and PCIe switches. We do not plot the NIC as in Fig. 6(a) because under that topology, the NIC has dedicated PCIe lanes and will not compete for the PCIe bandwidth with GPUs. The system architecture is modeled as a hierarchical graph $G=(V,E)$. Denote N as the set of leaf nodes (GPUs), S as the set of intermediate nodes (switches), C as the set of CPU nodes. $V=N\cup S\cup C$. Each edge $e(v_x,v_y)$ in E represents the bandwidth from vertex v_x to v_y, and we denote $t(v_x,v_y)$ as the amount of traffic sent from v_x to v_y. We further define p as the number of switches ($p\geq 2$), and n as the leaf nodes that each switch connects ($n\geq 2$).

We assume the following features of G: (1) Each edge in E is duplex and the bandwidth of both directions are equal. Denote $b(v_x,v_y)$ as the bandwidth of $e(v_x,v_y)$, then $b(v_x,v_y)=b(v_y,v_x)$; (2) We assume G is symmetric. The bandwidth at the same layer of the tree is equivalent. For example, $b(S_j,C_j)=b(S_k,C_k)$ and $b(N_x,S_j)=b(N_y,S_j)$ hold for any $j,k\in[0,p-1]$, $x,y\in[jn,(j+1)n-1]$; (3) The memory and QPI bandwidth is much higher than the PCIe links and is less likely to be the bottleneck. In the following, we only focus on the PCIe links.

The GPUs from N_0 to N_{pn-1} need to sum their data. We can either use *CPU-assisted aggregation* mentioned before, or use *brute-force copy* that needs each GPU to copy its entire data to C directly. In practice, the optimal solution should be a combination of these two strategies, depending on the value of $b(S_j,Cj)$ and $b(N_i,S_j)$. The intuition is that we apply brute-force copy on x of the data, and CPU-assisted aggregation on y of the data ($x+y=1$). Under certain x and y, the job completion time J can be minimized. We calculate the traffic of two links respectively. On $e(S_j,C_j)$, the traffic is composed of n times brute-force copy plus the traffic of CPU-assisted aggregation. On $e(N_i,C_j)$, the traffic is composed of one brute-force copy and the complete traffic of CPU-assisted aggregation.

$$t(S_j,C_j)=n*xM+\frac{yM}{n}*n=(nx+y)M \tag{8}$$

$$t(N_i,S_j)=xM+(\frac{2(n-1)}{n}+\frac{1}{n})yM=(\frac{2n-1}{n}y+x)M \tag{9}$$

Since J is determined by $J=max(\frac{t(N_i,S_j)}{b(N_i,S_j)},\frac{t(S_j,C_j)}{b(S_j,C_j)})$, the optimal J is highly related to the two bandwidth terms. On our own PCIe machines (Fig. 6(a)), we measure that both $b(N_i,S_j)$ and $b(S_j,C_j)$ are 13.1GB/s (105Gbps). Let M=1GB and $n=4$, combining Equation (8), (9) and $x+y=1$, we are trying to find a $x\in[0,1]$ such that $\arg\min_x J(x)=max(\frac{3x+1}{13.1},\frac{7-3x}{52.4})$. Solve it and we will get the optimal solution is $x^*=1/5$ and $J^*=0.129s$. This means the optimal solution works like this: each GPU applies brute-force copy on its $1/5$ data, and uses CPU-assisted aggregation for the rest $4/5$ data. Therefore, we have the following key conclusions:

CPU-assisted aggregation is near-optimal. When $x=0$, the solution is our CPU-assisted aggregation, and the job completion time is $J(0)=0.141s$. As calculated, the optimal time is $0.129s$. Thus, our strategy closely approximates the optimal solution, with 9% difference on performance. However, in practice, brute-force copy heavily stresses the CPU memory – any tensor that uses brute-force copy would consume $4\times$ CPU memory bandwidth compared with CPU-assisted aggregation. CPU memory does not really have $4\times$ bandwidth of PCIe links, especially for FP16 summation (Fig. 9(b). Consequently, we choose not to use brute-force copy at all and stick

to CPU-assisted aggregation.

CPU-assisted aggregation is better than ring-based all-reduce. We have the job completion time for ring-based all-reduce as $J_{ar} = \frac{2(np-1)M}{np*b_{bottleneck}}$. Similarly, for CPU-assisted aggregation we have $J_{ca} = \frac{M}{b(S_j,C_j)} * max(1, \frac{2n-1}{kn})$, where $k = \frac{b(N_i,S_j)}{b(S_j,C_j)}$. In our case, $k = 1$ and $b_{bottleneck} < b(S_j,C_j)$, so it is easy prove that $J_{ca} < J_{ar}$ always holds for any $n, p \geq 2$. For example, using the value from our PCIe machines, let $p = 2$, $n = 4$, $b_{bottleneck} = 80Gbps$ (bandwidth of memory copy that crosses PCIe switches) and $b(S_j,C_j) = 105Gbps$ we get that J_{ca} is 23.7% smaller than J_{ar}.

4.2.2 NVLink-based Topology

Fig. 8(a) shows the other machine model in our data center – a GPU machine with NVLinks. There are four PCIe switches, each connecting two GPU cards. The GPUs are also connected via NVLinks. The NVLinks give every GPU in total 1.2Tbps GPU-GPU bandwidth, much higher than the PCIe link. The NIC is connected to one of the PCIe switches.

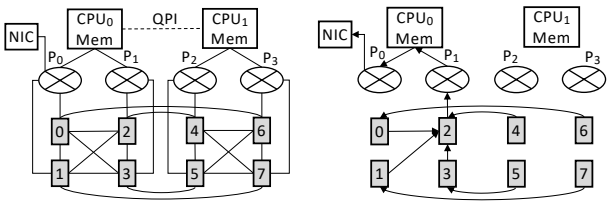
With NVLink, GPU-to-GPU communication can completely avoid consuming PCIe bandwidth. So, we no longer need CPU-assisted aggregation. However, we find that existing framework, including the most popular GPU all-reduce implementation NCCL (used by TensorFlow, PyTorch, MXNet and Horovod), is again sub-optimal.

The problem is that the topology is not symmetric considering the NIC, which is connected to only one (out of four) PCIe switch. The NIC and the two GPUs under the same PCIe switch have to compete for the PCIe bandwidth of $P_0 - CPU_0$. Remember that not only CS uses this PCIe bandwidth, but also the SS runs on this same GPU machine uses it! $P_0 - CPU_0$ again becomes the bottleneck in the whole communication.

Based on the analysis, we should leave as much $P_0 - CPU_0$ PCIe bandwidth as possible to the NIC during local aggregation. For this topology, BytePS uses reduce and broadcast instead of reduce-scatter and all-gather – tensors from all GPUs are first reduced to GPU_2 and the result is then copied to CPU_0 memory from GPU_2. Fig. 8(b) shows those steps. Later, when CS gets the aggregated results from SS, GPU_2 would copy the data into GPU memory and broadcast them to other GPUs. This way, we completely prevent GPUs from using the $P_0 - CPU_0$ bandwidth for communication, so the NIC can run to full 100Gbps bandwidth.

This approach seems to create traffic hotspots on GPU_2. However, NVLinks has much larger bandwidth than PCIe links, so inter-GPU communication is never the bottleneck even on the hotspots. Meanwhile, the $P_1 - CPU_0$ PCIe link used for GPU-CPU copy has approximately the same 100Gbps bandwidth as the NIC, so it is not a bottleneck either.

BytePS has achieved the optimal result – there is no intra-machine bandwidth bottleneck. Existing solutions like NCCL, unfortunately, tends to let GPUs use the $P_0 - CPU_0$ bottleneck link because of the proximity between GPU_0 and the NIC.



(a) NVLink-based topology (b) Outgoing data flow

Figure 8: NVLink-based machine topology and BytePS data flow. Only the outgoing direction is shown in the data flow figure.

Consequently, its communication performance is lower than our solution in the NVLink-based machines.

4.2.3 Discussion

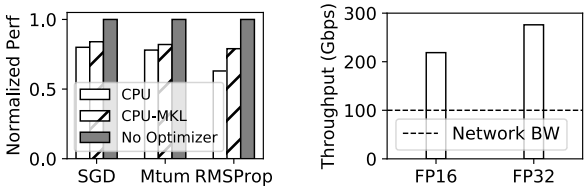
The solutions for PCIe-only and NVLink-based topology are quite different. This shows that there is no one-fit-all optimal solution. The intra-matchine communication must adapt to different internal topologies. Admittedly, there are certainly more topologies than the above two used in our environment. However, we believe that the above two are representative, since they are similar to the reference design recommended by server vendors [15] and NVIDIA [11], respectively.

Despite the difference, we summarize two principles – 1) always avoid direct GPU-to-GPU memory copy when the two GPUs are not under the same PCIe switch because it is slow in practice. 2) Always minimize traffic on the PCIe switch to CPU link that is shared by GPUs and NIC. We propose the following best practice procedure. Let S_n be the number of PCIe switches with GPUs and NIC, and S_g be the number of PCIe switches with only GPUs.

1. If $S_n > 0$ and $S_g > 0$, the topology is asymmetric like our NVLink-based topology. CS should use reduce and broadcast, with GPUs that are not competing with NICs as reduce or broadcast roots.
2. If $S_n = 0$ or $S_g = 0$, the topology is symmetric like our PCIe-only case. CS should use reduce-scatter and all-gather to balance traffic on all PCIe switches. CPU-assisted aggregation (§4.2.1) should be used if no NVLink.

Multi-NIC topology. Although the two specific topologies we discussed have only one NIC, the above principles can directly extend to multi-NIC topology – it only changes the value of S_n and S_g.

GPU-direct RDMA (GDR). GDR can potentially reduce the PCIe traffic. However, GDR requires the GPU and the RDMA NIC to be on the same PCIe switch, otherwise the throughput can be less than 50Gbps even with 100GbE NIC [12], which is also confirmed by our own measurements. Consequently, GDR does not benefit our settings – PCIe-only topology does not satisfy the requirement, and we already avoided any PCIe bottlenecks for NVLink-based topology. In addition, most clouds like AWS do not support GDR. Therefore, BytePS does not use GDR for now.



(a) Parameter update on different devices. (Mtum: Momentum [65])

(b) Throughput of CPU summation on different floating point tensors.

Figure 9: CPU is slow for optimizers but not for summation.

We can see that the optimal intra-machine communication strategy is tightly coupled with the internal topology. Building a profiler to automatically detect the topology, probe the bandwidth, and generate the best strategy is interesting future work.

5 Summation Service

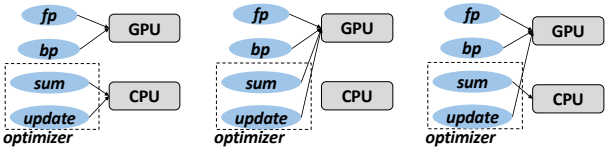
To get the optimal inter-machine communication time (§4.1), BytePS needs a module that can run on the CPU of every machine and communicate with CS. The question is, what is its role in the training algorithm? Our initial attempt was to follow the previous PS design [44], in which the PS processes are responsible for running the *optimizer*. The optimizer aggregates the gradients from all GPUs and updates the DNN model parameters using various optimizers.

The CPU bottleneck. Unfortunately, soon we found that the CPUs became a bottleneck in the system. We use an experiment to demonstrate this. We train the VGG16 DNN [63] using a typical non-colocated PS setting: using one Tesla V100 GPU machine and one CPU machine (Intel Xeon Platinum CPU, 32 cores with hyper-threading and Intel MKL [7]) connected by 100GbE Ethernet. The GPU machine runs the forward and backward propagation, and the CPU machine runs the optimizer using all the 32 CPU cores.

Fig. 9(a) shows that, even with 32 cores and MKL-enabled, running the optimizer on the CPU machine can slow down the end-to-end training speed. It means the CPU cannot match the network bandwidth and becomes a bottleneck (§6). As the optimizer algorithm gets more complicated (from simpler SGD to the more complicated RMSProp), the bottleneck effect becomes more severe.

The root cause. The CPU bottleneck is caused by the limited memory bandwidth. Popular optimizers such as Adam can easily exhaust the memory bandwidth of modern CPUs. For example, the peak transfer rate of a 6-channel DDR4-2666 memory setup is up to 1024 Gbps combining read and write [8]. It is easy to estimate that, for example, the Adam optimizer [42] requires more than 10x memory access (read+write) for applying every gradient update. Adding that 100Gbps NIC consumes 200 Gbps memory bandwidth (read+write), the 1024 Gbps memory bandwidth is simply not sufficient for Adam to process 100 Gbps gradient stream.

CPU is good at summation. The above experiment leads us to rethink the tasks placed on CPUs.The computation of an optimizer can be divided into two steps, *gradient summation* and *parameter update*, as Fig. 10 shows.



(a) PS (b) All-reduce (c) BytePS

Figure 10: Component placement comparison between all-reduce, PS and BytePS.

Fortunately, modern x86 CPUs are good at summation thanks to the highly optimized AVX instructions [47]. In Fig. 9(b), we show the summation throughput on the same CPUs as above, using synthetic floating point tensors. The throughput is more than 200Gbps for both FP16 and FP32 precision, higher than the 100Gbps NIC bandwidth. Therefore, summation on CPU will not be a bottleneck.

BytePS's solution. Based on these observations, BytePS decouples the two steps of optimizer. We move the computation-intensive parameter update to GPUs and places only summation on CPUs – this is why we name the CPU module Summation Service (SS). SS not only prevents the CPU from being the bottleneck, but also largely reduces the CPU overhead. With carefully implementation using AVX and OpenMP, SS only consumes fewer than 3 CPU cores when it runs at 100Gbps throughput. Fig. 10 gives a high-level comparison over PS, all-reduce and BytePS on how they place different components in DNN training onto GPU and CPU resources.

Since Summation Service moves parameter update to GPU machines, all the GPU machines need to perform the same parameter update calculation, whereas parameter update needs to be done only once in traditional PS. BytePS hence uses more computation cycles for parameter update than PS. This is a tradeoff we made willingly, to accelerate end-to-end training speed. We define SS overhead ratio as the FLOPs for parameter update over the sum of FP and BP FLOPS. The ratio is 138 MFLOPs / 32 GFLOPs, 26 MFLOPs / 7.8 GFLOPs, 387 MFLOPs / 494 GFLOPs for VGG-16, ResNet-50, BERT-large using SGD as the optimizer, all are less than 0.5%. The introduced overhead is negligible, compared to the training speedup (Fig. 9(a)). The above ratio definition assumes batch size of 1. DNN training typically uses batch size of tens or hundreds. Parameter update is done once per batch, hence the additional overhead is even smaller in practice.

We note that Horovod [60] has the option to move gradient aggregation to CPUs by first copying the tensors to CPU memory and then performing CPU-only all-reduce. Since it still only relies on the CPUs and bandwidth on GPU machines, it does not provide communication-wise advantages compared with directly all-reduce on GPUs. BytePS is different: it leverages *additional* CPU machines for gradient summation, while keeps parameter update on GPUs.

Support asynchronous training. Although separating the

summation and update brings us performance benefits, it breaks an important feature of the original PS: the support of asynchronous training like Asynchronous Parallel [25]. Asynchronous Parallel relies on the PS processes keeping the most updated model parameters, which is not directly compatible with the design of SS. To bridge this gap, we re-design a new workflow that can enable asynchronous training with SS, as shown in Fig. 11(b). In short, GPU updates parameters and computes the *delta parameters* first. CS sends them and receives *latest parameters*. SS keeps adding delta parameters to the latest parameters. Next, we prove that this new training workflow is equivalent to Asynchronous Parallel in terms of algorithm convergence.

Theorem 2. *The asynchronous algorithm for BytePS is equivalent to Asynchronous Parallel [25].*

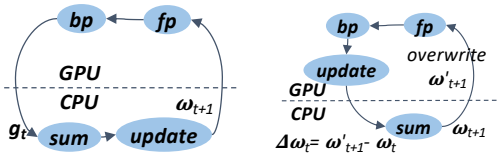
Proof. Consider one SS connected with n CSs. We say a CS stores the local model parameters, and a SS holds the latest version of parameters. The high level idea of our proof is to show that our algorithm generates identical state (*i.e.*, same parameter for the SS module and n CS modules) with Asynchronous Parallel, given the same communication order (push and pull order). We use f as a general representation of the optimizer. The optimizations thus can be represented as $w \leftarrow w + f(g_{i,t})$, where $g_{i,t}$ represents the gradients of CS_i ($i \in [0, n-1]$) at iteration t ($t \in [1, T]$). Denote w_{ps} and w_{byteps} as the parameter in PS and BytePS, respectively. And denote $w_{i,t}$ as the parameter on each $worker_i$ (for PS) or CS (for BytePS) at iteration t. The parameter is initiated to w_0 for all CSs and the SS. After T iterations, we can obtain the updated parameter as:

$$w_{ps} = w_0 + \sum_{t=1}^{T} \sum_{i=0}^{n-1} f(g_{i,t}) \tag{10}$$

$$w_{byteps} = w_0 + \sum_{t=1}^{T} \sum_{i=0}^{n-1} \Delta w_{i,t} \tag{11}$$

Next, we use induction to prove that $\Delta w_{i,t} = f(g_{i,t})$ holds for any i and t. (1) *Base case* $t = 1$: Given initial parameter w_0, we obtain the gradient $g_{i,1}$ from w_0. In Parameter Server, $worker_i$ pushes $g_{i,1}$ to the server and get updated as $w_{ps,1} = w_0 + f(g_{i,1})$. In BytePS, CS_i pushes $f(g_{i,1})$ to SS and get updated as $w_{byteps,1} = w_0 + f(g_{i,1})$. So $\Delta w_{i,t} = f(g_{i,t})$ holds for $t = 1$. Meanwhile, the parameter on $worker_i$ or CS_i is the same on both architectures after receiving the response from the server or SS. (2) *Inductive step*: If the lemma we want to prove holds for $t = k (k \geq 1)$, the gradient $g_{i,k+1}$ is computed from the same w_k. Similar to the base case, we obtain $w_{ps,k+1} = w_k + f(g_{i,k+1})$ and $w_{byteps,k+1} = w_k + f(g_{i,k+1})$. So $\Delta w_{i,t} = f(g_{i,t})$ holds for $t = k+1$. By the principle of induction, $\Delta w_{i,t} = f(g_{i,t})$ holds for all $t \in \mathbb{N}$.

Return to (10) and (11). Since $\Delta w_{i,t} = f(g_{i,t})$ holds for any i and t, we get $w_{ps} = w_{byteps}$. This completes the proof because the parameter of our algorithm and Asynchronous Parallel are equivalent after any T batches. □



(a) PS-async (b) BytePS-async

Figure 11: Asynchronous training workflow comparison between PS and BytePS. g is the gradients. ω is the parameters.

6 Implementation

While the core of BytePS is generic for any training framework, BytePS also implements plugins for TensorFlow, PyTorch and MXNet, for user-friendliness. The core is implemented in C++, while the framework plugins contain both C++ and Python. In total, BytePS consists of about 7.8K lines of Python code, and 10K lines of C++ code. As a major online service provider, we have deployed BytePS internally. BytePS has also been open-sourced [4] and attracted thousands of users.

6.1 Multi-Stage Pipeline

A common way to speed up a multi-step procedure is to build a multi-stage pipeline that overlaps the processing time of each step. We incorporated the idea of tensor partition and pipelining from prior work [34, 55]. For example, for PCIe-only topology, CS has six steps. It maps to a 6-stage pipeline in BytePS runtime. We implement BytePS to be flexible in constructing the pipeline without recompiling. Each stage in the pipeline is implemented as an independent thread with a priority queue of tensors. The priority is assigned similar to [34,55]. As analyzed in §4.1.1, large tensors are partitioned to multiple smaller tensors no more than 4MB. Next, each small tensor is enqueued to the first queue and moves towards the next queue once a stage finishes processing it, until it is dequeued from the last one.

6.2 Address RDMA Performance Issues

For inter-machine communication, we use RDMA RoCEv2. Each machine has one 100GbE NIC, and the RDMA network provides full bisection bandwidth. To get the full benefit of RDMA, we have gone through a full design and debug journey which we share as follows.

RDMA Memory Management. To improve the performance, we aim to avoid unnecessary memory copies [72] and achieve zero-copy on CPU memory. BytePS is based on RDMA WRITE because it is the most performant among common RDMA verbs [39]. Conventional one-sided RDMA operations (WRITE and READ) require at least two round-trips: getting the remote address, and writing (reading) the value to (from) that address [39, 40, 50, 70]. We optimize the process by leveraging the fact that DNN training always sends the same set of tensors in every iteration. Only at the

Table 2: BytePS throughput with a pair of CPU machine and GPU machine running microbenchmark.

Solution	baseline	+shm	+shm +aligned	all
Throughput in Gbps	41	52 (1.27x)	76 (1.85x)	89 (2.17x)

first iteration, BytePS initializes all the required tensors, register the buffer with RDMA NIC and exchange all the remote addresses. Then BytePS stores the remote buffer information and reuse it directly in the rest iterations.

Address Slow Receiver Symptom. We also run into the slow receiver symptom as reported in [30] – the NICs are sending out many PFCs into the network. Those excessive PFCs slow down tensor transmission can cause collateral damage to other traffic. Here we report several additional causes of such symptom and how we address them.

Our first finding is that internal RDMA loopback traffic can cause internal incast, and push the NIC to generate PFC. BytePS runs both CS and SS on each GPU machine. The traffic between them, which we call loopback traffic, does not consume NIC's external Ethernet bandwidth, but does consume internal CPU-NIC PCIe bandwidth. Initially, we did not add any special design – we stuck to RDMA verbs [9] for loopback traffic and thought the NIC DMA can handle it. However, we realize that it creates a 2:1 incast on the NIC, with RX and loopback as two ingress ports and the DMA to memory engine as one egress port!

To solve it, we implement a shared memory (*shm*) data path. When CS detects that SS is on the same machine as itself, CS simply notifies SS that the data is in shared memory. After SS finishes summation, SS copies the results from its own buffer back to CS's shared memory. Consequently, the loopback RDMA traffic is eliminated.

Our Second finding is that we need to use *page-aligned* memory for RDMA. Otherwise PFCs may be triggered. Our hypothesis is that hardware DMA aligns the transfer unit to the page size (*e.g.,* 4096 bytes). Therefore, using a page-aligned address is more friendly to DMA engine as it reduces the number of pages needed to be written.

Our third finding is that the RDMA NIC RX performance can be impacted by how the concurrent send is implemented! In the end, we not only use page-aligned memory, but also enforce only one scatter-gather entry (*sge*) per RDMA WRITE on the sender side.[7]

After all the optimization, BytePS implementation can run as expected. Table 2 shows the performance improvement after each of the above three optimizations is applied. The NIC generates negligible PFCs.

As we have discussed in §4.1, BytePS creates many many-to-one communication patterns in the network. Many-to-one is well-known for creating incast and packet loss in TCP/IP network [66]. But BytePS uses RDMA/RoCEv2 which depends on a lossless fabric and DCQCN [75] for congestion control. We do not observe incast issue in BytePS.

[7] In the whole process, we contacted with the NIC vendor and had lengthy discussion with their software and hardware experts. As of writing, we have not got the official root cause of the last two problems.

6.3 BytePS Usage

BytePS [4] is easy to use. We provide Python interfaces that are almost identical to Horovod, PyTorch native API and TensorFlow native API. Users can choose either of them and migrate to BytePS with minimal efforts. For example, for a Horovod-MNIST example [19], we only need to change one line of Python code, from *"import horovod"* to *"import byteps"*. In fact, we are able to convert most of our internal Horovod-based training tasks to BytePS automatically.

7 Evaluation

In this section, we show that BytePS not only achieves optimal communication performance in microbenchmarks, but also significantly accelerate training jobs in production environment. We list a few highlights regarding the high fidelity of the results.

- All resources used are allocated by the scheduler of production clusters. The scheduler uses non-preemptive resource scheduling – once a training job is scheduled, it will have a fixed number of CPU machines that will not change. Even the most large-scale tasks we show use $< 5\%$ GPUs of a cluster that runs many production tasks.
- We use large training batch sizes. Smaller batch sizes mean less GPU memory consumption but more communication, so the end-to-end improvement will be more evident. However, all our tasks use almost full GPU memory, so the speedup numbers against all-reduce and PS are the *lower bound* of BytePS.
- Although we cannot disclose any specific models that are used internally, the tasks and DNN model structures shown are highly representative of production workloads. The code is also available publicly for reproducibility [5].
- We compare BytePS with the state-of-the-art PS and all-reduce implementation without modification. For example, we do not apply the RDMA optimizations mentioned in §6.2 on native-PS and all-reduce.

The cluster we use has a RoCEv2 network with full bisection bandwidth. All the machines have one 100GbE NIC. We note that TensorFlow, PyTorch and MXNet can overlap the DNN computation and communication [34, 55], thus even a small improvement in end-to-end performance can indicate a large improvement in communication.

7.1 Inter-machine Microbenchmarks

First, we use microbenchmarks to show the pure inter-machine communication performance of different architectures. We allocate eight 1-GPU machines from the cluster scheduler. We run a dummy task in which all GPU workers

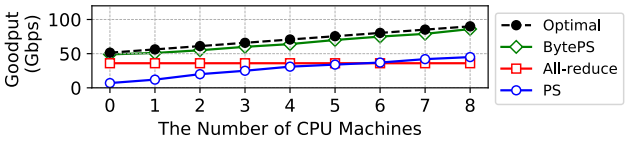
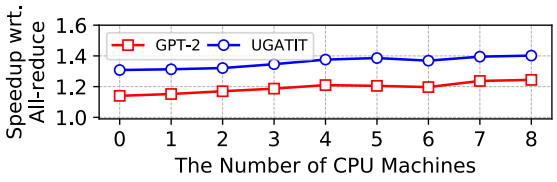
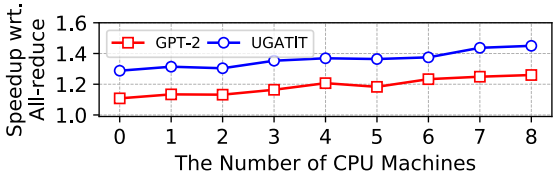


Figure 12: Communication goodput of 8× 1-GPU machines with varying number of additional CPU machines. The point-to-point RDMA goodput is ≈ 90*Gbps* in our network, so we plot the "optimal" line based on $B = 90Gbps$ and the analysis in §4.1.



(a) PCIe-only GPU machines



(b) NVLink-based GPU machines

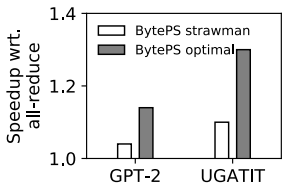
Figure 13: End-to-end performance with different number of CPU machines. The training is run with PyTorch on 8 GPU machines each with 8 GPUs. Each CPU machine uses < 4 cores.

just keep reducing large tensors on GPU and record the communication goodput. We verify that no other distributed job is placed on the same physical machines.

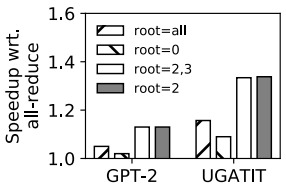
Fig. 12 shows that BytePS performance is very close to the theoretical optimum (§4.1), with 1-9% difference for different number of CPU machines. All-reduce, as expected, is close to the optimal only if there is no additional CPU machine, while remain the same even if there are CPU machines. The (MXNet) PS does not run optimizer in this case, but is mainly bottlenecked by issues described in §6.2. In practice, if PS runs DNN optimizer algorithms, the performance will be worse than all-reduce even with $k = n$ CPU machines (Fig. 4). In contrast, because of the Summation Service design, BytePS would not be affected in real training tasks shown below.

7.2 Leverage CPU Machines

Next, we show that BytePS can indeed leverage different numbers of CPU machines to speed up training. In Fig. 13, we use 8 GPU machines, each with 8 Tesla V100 32GB GPUs, and is either PCIe-only or NVLink-based topology. We vary the number of CPU machines from 0 to 8. We compare BytePS end-to-end training performance against state-of-the-art all-reduce implementation (Horovod 0.19 and NCCL 2.5.7) as the baseline. We test two DNN models, UGATIT GAN [41] (one of the most popular models for image generation) and GPT-2 [57] (one of the most popular NLP models for text generation), both implemented in PyTorch. The per GPU batch



(a) PCIe-only GPU machines



(b) NVLink-based GPU machines

Figure 14: Topology-aware intra-machine communication. The training is run with PyTorch on 8 GPU machines each with 8 GPUs and no additional CPU machine.

size is 2 images for UGATIT, and 80 tokens for GPT-2. We will evaluate more models, frameworks and machines in §7.4.

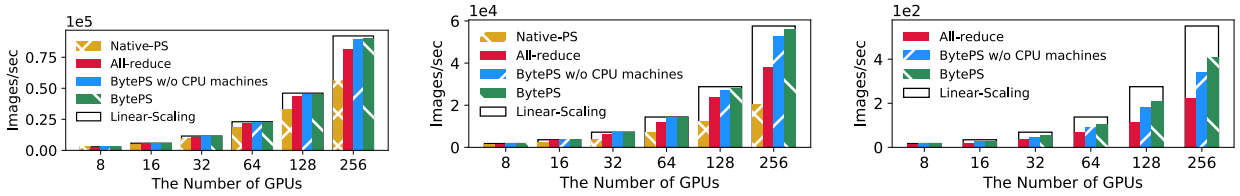
Fig. 13 shows that, with more CPU machines, BytePS can run faster – up to 20% than without CPU machines. The SS on each CPU machine only consumes no more than 4 CPU cores. It is usually easy for our scheduler to find sufficient CPUs that are on machines running non-distributed jobs. It is free (or << 10% costs compared with the expensive GPUs) speedup for the cluster. Compared with all-reduce, BytePS is consistently faster in any cases and can be up to 45% faster in the best case. On NVLink-based GPU machines, the speedup is higher because the communication bottleneck is more on the network instead of PCIe links. Finally, models have different speedup due to different model sizes and FLOPs. In the examples we show, GAN is more communication intensive, so the end-to-end gain of BytePS is larger.

7.3 Adapt to Intra-machine Topology

Next, we show the benefits of BytePS intra-machine communication strategy. The software and hardware configurations are the same as in §7.2. To better compare with the all-reduce baseline, we run the jobs without any CPU machines. Thus, BytePS does not take any advantages explained in §7.2. For PCIe-only GPU machines (Fig. 14(a)), we run BytePS with 1) strawman strategy, the same as common all-reduce or PS and 2) the optimal solution in §5. We see that the optimal intra-machine solution has up to 20% gain as well.

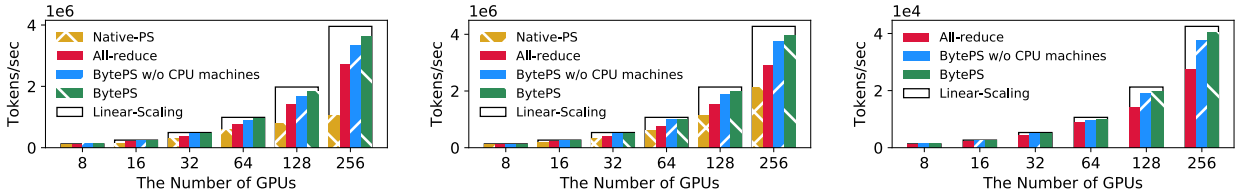
For NVLink-based GPU machines (Fig. 14(b)), we use different sets of GPUs as the local reduce roots. BytePS's optimal solution, as explained in §4.2.2, is $root = 2$. $root = 2,3$ means CS chooses GPU 2 and 3 as the reduce root in a round robin manner. It has almost the same performance because GPU 3 is not competing for PCIe bandwidth with the NIC, either. It is an alternatively optimal solution. However, $root = all$ has poorer performance. Communication-wise, it is equivalent to Horovod's hierarchical mode. $root = 0$ is the worst because it competes hardest with the NIC. Unfortunately, it is equivalent to Horovod's normal mode (plain NCCL all-reduce).

One thing to note is that even without any optimization, BytePS still outperforms all-reduce. We discuss this in §8.



(a) TensorFlow, ResNet-50, batch=256 images (b) MXNet, VGG-16, batch=96 images (c) PyTorch, UGATIT, batch=2 images

Figure 15: Computer Vision models. The batch sizes are per GPU.



(a) TensorFlow, Transformer, batch=3072 tokens (b) MXNet, BERT-Large, batch=8192 tokens (c) PyTorch, GPT-2, batch=80 tokens

Figure 16: NLP models. The batch sizes are per GPU.

7.4 Scalability

To demonstrate BytePS's performance at different scales, we run six different training jobs using 8 to 256 Tesla V100 32GB GPUs, *i.e.,* 1 GPU machine to 32 GPU machines. Due to the constraint of free resources, we only use NVLink-based GPU machines. The six different jobs cover three frameworks, TensorFlow, PyTorch and MXNet. We have introduced two of the models, UGATIT and GPT-2 in §7.2. The rest four models are ResNet-50 [32], VGG-16 [63] (two of the most popular models for image classification and extraction), Transformer [67] (one of the most popular models for machine translation) and BERT [26] (one of the most popular models for natural language understanding). We take the official implementation of these models and slightly modify them (no more than 20 lines of code) to use PS, all-reduce and BytePS, respectively.

For BytePS, we evaluate its performance with and without CPU machines. When there are CPU machines, the number of CPU machines is equal to GPU machines. For all-reduce, we use Horovod with NCCL for all cases. For PS, we show the native implementation from TensorFlow and MXNet with RDMA support enabled. PS uses the same resources as BytePS with CPU machines. PyTorch does not have official PS implementation, so it does not have PS results. We also provide the speed of linear scaling as the upper bound. We use trained images per second as the speed metric for computer vision models, and tokens per second for NLP models.

Fig. 15 and Fig. 16 show very consistent results – BytePS with CPU machines is always the best and BytePS without CPU machines is the second. The native PS of both TensorFlow or MXNet are always the poorest. All-reduce always has a clear advantage over PS, but is inferior to BytePS. When training with 256 GPUs, the speedup of BytePS over all-reduce is 10% to 84% with CPU machines, and 9% to 53% without CPU machines. From 8 GPUs to 256 GPUs, the speedup becomes larger. We expect that with even more GPUs, BytePS will have even larger advantage.

We see that models have different system scalability,[8] which is determined by the model sizes and FLOPs. The most scalable model is ResNet-50. BytePS achieves 97.5% scaling efficiency with 256 GPUs. All-reduce also performs well, achieving 88% scaling efficiency. It is not surprising that prior work is fond of training ResNet at large scale [49, 73] with all-reduce. Nevertheless, other models are more challenging, with UGATIT as the least scalable one. Even BytePS only achieves 74% scaling efficiency. For such communication intensive models, BytePS has the most gain over all-reduce (84% with 256 GPUs). Despite UGATIT, BytePS has at least 91.6% scaling factor for the rest five 256-GPU training jobs.

We analyze the breakdown of performance improvement by comparing native-PS and BytePS, since they both use the same number of additional CPU machines. For example, BytePS outperforms native-PS by 52% with 256 GPUs on VGG-16 (Fig. 15(b)). Among the 52% improvement, we find that 19% comes from optimal communication design (intra-server), 18% comes from Summation Service, and the rest 15% comes from better implementation mentioned in §6.

8 Observations and Discussion

In this section, we share several of our observations and discussions, with the aim to inspire future research.

BytePS outperforms all-reduce even without extra CPU machines. Theoretically, the communication time is the same for all-reduce and BytePS when no additional CPU machines are available (§4.1). In practice, we observe that BytePS still outperforms all-reduce significantly in this case. One reason is that BytePS has a better intra-machine communication strategy than all-reduce. However, even without intra-machine optimization, BytePS still outperforms all-reduce (see Fig. 14 in §7). We hypothesize that BytePS has the advantage of

[8] We focus on system scalability and do not discuss algorithm scalability, *i.e.* the hyperparameter tuning and convergence speed with more GPUs.

allowing more "asynchronicity" than all-reduce. All-reduce usually requires additional out-of-band synchronization to ensure the consistent order across nodes, while BytePS does not have this overhead. However, to analyze it, we need a distributed profiler that can build the complete timeline of the execution and communication across all nodes in distributed training.

GPU cluster scheduler should consider dynamic CPU resources. By leveraging additional CPU machines, BytePS can speedup DNN training. Since BytePS can adapt to any number of CPU machines, it enables elasticity – the cluster scheduler can scale in or out CPU machines for existing jobs based on real time conditions. Most existing schedulers keep the number of GPUs of a job *static* because of convergence problems [16, 74]. Fortunately, the number of CPU machines in BytePS only impacts system performance but not model convergence. We plan to add elasticity support to BytePS, which will enable BytePS to *dynamically* schedule CPU resources during the training process.

Model-parallelism support. BytePS can accelerate the communication when reducing tensors across GPUs. Some model parallelism methods, such as Megatron-LM [62] and Mesh-TensorFlow [61], also rely on the all-reduce primitive for communication. Therefore, BytePS can also accelerate them by replacing the all-reduce operations.

9 Related Work

Acceleration of computation: To accelerate the forward propagation and backward propagation, the community has worked out many advanced compilers and libraries, including cuDNN [10], MKL [7], TVM [23], XLA [17], Astra [64] and other computation graph optimization, *e.g.,* Tensor Fusion [14] and graph substitution [37]. They focus on speeding up DNN computation. They are complementary to and can be used with BytePS.

Acceleration of communication: There are several directions for accelerating communication: (1) *Gradient compression* [21, 45] is proposed to reduce the communication traffic, *i.e.*, using half precision for gradient transmission, at the cost of potential degradation of accuracy. (2) *Communication scheduling and pipelining*: Recent work explores to better overlap the computation and communication by priority-based scheduling and tensor partition [31, 34, 55]. The ideas are that tensor partition enables simultaneous bidirectional communication, and that during communication, the former layers have higher priority because they are needed sooner for FP of the next iteration. Those ideas are complementary to BytePS, and they have been integrated into our implementation. Pipedream [51] adds parallelism between multiple batches. BytePS can also potentially accelerate its data parallel stages.

Hierarchical all-reduce: Some work proposes to leverage the hierarchical topology [24, 49] during all-reduce, in order to minimize the traffic at bottleneck links. However, they still rely on the assumption that resources are homogeneous while overlooking CPU resources. BytePS can outperform them by leveraging the heterogeneous resources. In fact, the latest NCCL includes hierarchical, tree-based all-reduce, which does not differ much from the results in §7.

Intra-machine optimization: Blink [68] also optimizes multiple GPU communication inside a single machine, by leveraging hybrid transfers on NVLinks and PCIe links. However, Blink does not optimize the distributed training cases, where the main communication bottleneck is the NIC and its PCIe connection instead of the much faster NVLinks. BytePS carefully schedules the intra-machine traffic to utilize the bottleneck bandwidth better – the NIC bandwidth. Our intra-machine design also considers the PCIe bandwidth consumed by the NIC, while Blink is only focused on GPU's PCIe connections.

New hardware chips or architecture for accelerating DNN training: Recently, there are many new chips, like TPU [38] and Habana [6], that are specifically designed for DNN training. In fact, the design of BytePS is not GPU-specific, and should apply to them as long as they are also PCIe devices. Some also propose using InfiniBand switch ASIC [28] to accelerate all-reduce, or using P4 switches [58, 59] to accelerate PS. E3 [46] leverages SmartNICs to accelerate network applications, and can potentially benefit BytePS by offloading the gradient summation from CPUs to SmartNICs. PHub [48] proposes a rack-scale hardware architecture with customized network configurations, *e.g.,* 10 NICs on one server. BytePS focuses on using generally available CPU and GPU servers in commodity data centers.

10 Conclusion

BytePS is a unified distributed DNN training acceleration system that achieves optimal communication efficiency in heterogeneous GPU/CPU clusters. BytePS handles cases with varying number of CPU machines and makes traditional all-reduce and PS as two special cases of its framework. To further accelerate DNN training, BytePS proposes Summation Service and splits a DNN optimizer into two parts: gradient summation and parameter update. It keeps the CPU-friendly part, gradient summation, in CPUs, and moves parameter update, which is more computation heavy, to GPUs. We have implemented BytePS and addressed numerous implementation issues, including those that affect RDMA performance. BytePS has been deployed, extensively used and open sourced [4]. Multiple external projects have been developed based on it. The Artifact Appendix to reproduce the evaluation is at [3].

11 Acknowledgement

We thank our shepherd Rachit Agarwal and the anonymous reviewers for their valuable comments and suggestions. Yimin Jiang and Yong Cui are supported by NSFC (No. 61872211), National Key RD Program of China (No. 2018YFB1800303).

References

[1] A Light-weight Parameter Server Interface. https://github.com/dmlc/ps-lite.

[2] Amazon EC2 P3 Instances. https://aws.amazon.com/ec2/instance-types/p3/.

[3] Artifact Appendix. https://github.com/byteps/examples/blob/master/osdi20ae.pdf.

[4] BytePS. https://github.com/bytedance/byteps.

[5] Evaluation Code. https://github.com/byteps/examples.

[6] Habana. https://habana.ai/.

[7] Intel MKL. https://software.intel.com/en-us/mkl.

[8] Intel Xeon Platinum 8168 Processor. https://ark.intel.com/content/www/us/en/ark/products/120504/intel-xeon-platinum-8168-processor-33m-cache-2-70-ghz.html.

[9] Libibverbs. https://www.rdmamojo.com/2012/05/18/libibverbs/.

[10] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[11] NVIDIA DGX-1. https://www.nvidia.com/data-center/dgx-1/.

[12] NVIDIA GPU Direct RDMA Benchmark. https://devblogs.nvidia.com/benchmarking-gpudirect-rdma-on-modern-server-platforms/.

[13] NVIDIA NCCL. https://developer.nvidia.com/nccl.

[14] NVIDIA TensorRT Inference Library. https://devblogs.nvidia.com/deploying-deep-learning-nvidia-tensorrt/.

[15] Supermicro PCIe Root Architectures for GPU Systems. https://www.supermicro.org.cn/products/system/4U/4029/PCIe-Root-Architecture.cfm.

[16] Train ImageNet in 18 Minutes. https://www.fast.ai/2018/08/10/fastai-diu-imagenet/.

[17] XLA. https://www.tensorflow.org/xla.

[18] Amazon EC2 Pricing on demand. https://aws.amazon.com/ec2/pricing/on-demand/, 2019.

[19] TensorFlow MNIST Example with Horovod. https://github.com/horovod/horovod/blob/master/examples/tensorflow_mnist.py, 2020.

[20] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A System for Large-Scale Machine Learning. In *OSDI 2016*.

[21] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive Residual Gradient Compression for Data-parallel Distributed Training. In *AAAI 2018*.

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *LearningSys 2015*.

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI 2018*.

[24] Minsik Cho, Ulrich Finkler, and David Kung. BlueConnect: Novel Hierarchical All-Reduce on Multi-tired Network for Deep Learning. In *SysML 2019*.

[25] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large Scale Distributed Deep Networks. In *NIPS 2012*.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[27] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training Imagenet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[28] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *COMHPC 2016*.

[29] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI 2019*.

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

[31] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *SysML 2019*.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR 2016*.

[33] Geoffrey Hinton, li Deng, Dong Yu, George Dahl, Abdelrahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Phuongtrang Nguyen, Tara Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *Signal Processing Magazine, IEEE*, 2012.

[34] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In *SysML 2019*.

[35] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC 2019*.

[36] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly Scalable Deep Learning Training System with Mixed-precision: Training Imagenet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[37] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *SOSP 2019*.

[38] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA 2017*.

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

[40] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for Key-value Services. In *SIGCOMM 2014*.

[41] Junho Kim, Minjae Kim, Hyeonwoo Kang, and Kwanghee Lee. U-GAT-IT: Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization for Image-to-Image Translation. *arXiv preprint arXiv:1907.10830*, 2019.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR*, 2015.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS 2012*.

[44] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI 2014*.

[45] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv preprint arXiv:1712.01887*, 2017.

[46] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *ATC 2019*.

[47] Chris Lomont. Introduction to Intel Advanced Vector Extensions. *Intel white paper*, 23, 2011.

[48] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter Hub: A Rack-scale Parameter Server for Distributed Deep Neural Network Training. In *SoCC 2018*.

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

[50] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *ATC 2013*.

[51] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP 2019*.

[52] Tony Paikeday. Steel for the AI Age: DGX SuperPOD Reaches New Heights with NVIDIA DGX A100. https://blogs.nvidia.com/blog/2020/05/14/dgx-superpod-a100/, May 2020.

[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NIPS 2019*.

[54] Pitch Patarasuk and Xin Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 2009.

[55] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP 2019*.

[56] Raul Puri, Robert Kirby, Nikolai Yakovenko, and Bryan Catanzaro. Large Scale Language Modeling: Converging on 40GB of Text in Four Hours. In *SBAC-PAD 2018*.

[57] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 2019.

[58] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network Computation Is A Dumb Idea Whose Time Has Come. In *HotNets 2017*.

[59] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-network Aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[60] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *CoRR*, 2018.

[61] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. *arXiv preprint arXiv:1811.02084*, 2018.

[62] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv: 1909.08053*, 2019.

[63] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[64] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. Astra: Exploiting Predictability to Optimize Deep Learning. In *ASPLOS 2019*.

[65] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. In *ICML 2013*.

[66] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM 2009*.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *NIPS 2017*.

[68] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys 2020*.

[69] Yuxuan Wang, R. J. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc V. Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. Tacotron: A Fully End-to-End Text-To-Speech Synthesis Model. *CoRR*, abs/1703.10135, 2017.

[70] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *OSDI 2018*.

[71] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI 2018*.

[72] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards Zero Copy Dataflows Using RDMA. In *SIGCOMM 2017 Posters and Demos*.

[73] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image Classification at Supercomputer Scale. *arXiv preprint arXiv:1811.06992*, 2018.

[74] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. Reducing BERT Pre-Training Time from 3 Days to 76 Minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[75] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM 2015*.

[76] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized Stochastic Gradient Descent. In *NIPS 2010*.



Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads

Deepak Narayanan[†*], Keshav Santhanam[†*], Fiodar Kazhamiaka[†], Amar Phanishayee[⋆], Matei Zaharia[†]
[⋆]*Microsoft Research* [†]*Stanford University*

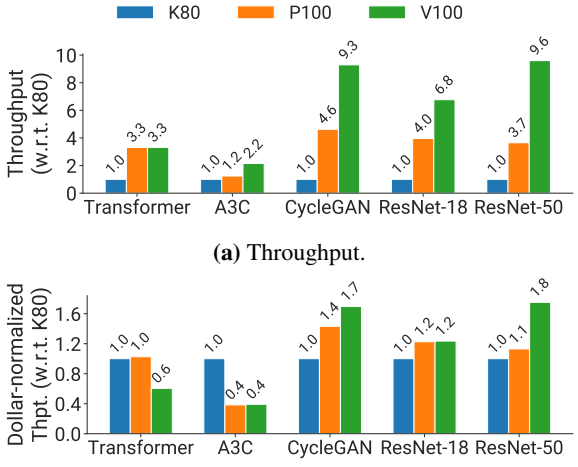
Abstract

Specialized accelerators such as GPUs, TPUs, FPGAs, and custom ASICs have been increasingly deployed to train deep learning models. These accelerators exhibit heterogeneous performance behavior across model architectures. Existing schedulers for clusters of accelerators, which are used to arbitrate these expensive training resources across many users, have shown how to optimize for various multi-job, multi-user objectives, like fairness and makespan. Unfortunately, existing schedulers largely do not consider performance heterogeneity. In this paper, we propose Gavel, a heterogeneity-aware scheduler that systematically generalizes a wide range of existing scheduling policies. Gavel expresses these policies as optimization problems and then systematically transforms these problems into heterogeneity-aware versions using an abstraction we call effective throughput. Gavel then uses a round-based scheduling mechanism to ensure jobs receive their ideal allocation given the target scheduling policy. Gavel's heterogeneity-aware policies allow a heterogeneous cluster to sustain higher input load, and improve end objectives such as makespan and average job completion time by 1.4× and 3.5× compared to heterogeneity-agnostic policies.

1 Introduction

As Moore's law comes to an end, specialized accelerators such as GPUs, TPUs, FPGAs, and other domain-specific architectures have emerged as an alternative to more general-purpose CPUs. These accelerators have been deployed to great effect [25, 35] to train state-of-the-art deep neural network (DNN) models for many domains, including language, image and video [14, 30, 31, 51, 55].

Consequently, users today must choose from a wide variety of accelerators to train their DNN models. For example, public cloud users can rent several generations of NVIDIA GPUs and Google TPUs from cloud providers [1–3]. Even organizations with private clusters have accumulated different accelerator types over time [34]; anecdotally, our research group has NVIDIA Titan V, Titan X, and P100 GPUs in its private cluster. Resources in these multi-tenant settings are typically arbitrated by a scheduler. GPU cluster schedulers such as Themis [40], Tiresias [28], AlloX [37], and Gandiva [58] thus need to decide how to allocate diverse resources to many users while implementing complex cluster-wide *scheduling policies*, optimizing objectives such as fairness or makespan. Unfortunately, choosing the most effective accelerator types in this context is difficult for three reasons:



(a) Throughput.

(b) Dollar-normalized.

Figure 1: Throughputs and dollar-normalized throughputs of training for various ML models. Dollar-normalized throughputs are computed by dividing the corresponding throughput by the relevant GCP on-demand price, The magnitude of speedup across GPU generations varies significantly across models.

Performance Heterogeneity. Commonly used models show heterogeneous performance behavior across accelerator types due to various architectural differences. For example, Figure 1a shows that a ResNet-50 model sees a nearly 10× speedup from an NVIDIA V100 GPU compared to a K80 GPU, while an A3C Deep Reinforcement Learning model only sees a 2× speedup. However, as shown in Figure 1b, the V100 is no longer the optimal choice for all models when we consider the number of samples trained per dollar – for many models, the older P100 GPU is competitive or cheaper on a per-dollar basis. Some scheduling policies can also benefit from splitting a job between *multiple* resource types: for example, minimizing a job's cost subject to a latency SLO (e.g., complete a job in 10 hours) might involve using a cheaper accelerator to begin training and then switching to a faster, more expensive device to meet the SLO. Thus, for even simple *single-job* settings, the choice of accelerator type is non-trivial and depends on *both* the job and the policy. This gets more complicated in *multi-job* settings as granting all jobs their

[*]Work done in part as interns at Microsoft Research.

preferred accelerator simultaneously might not be possible. Existing schedulers like Gandiva, Tiresias, and Themis do not consider this heterogeneous performance behavior.

Generality across Policies. Cluster operators might want to implement different scheduling policies based on their business goals, such as optimizing for time to complete a set of batch jobs (makespan), fairness for ad-hoc jobs, or more sophisticated *hierarchical* policies that divide resources among high-level entities (e.g., departments) using one policy, and then individual jobs within the entity using another [34]. In data analytics clusters, many job schedulers have support for hierarchical allocation policies [6, 7, 12, 59] already. The two recently proposed GPU schedulers that do consider heterogeneous resources, AlloX [37] and Gandiva$_{\text{fair}}$ [18], optimize for a single scheduling objective, and tightly couple their scheduling mechanism to that objective (e.g., max-min fairness). Thus, they cannot easily support the more sophisticated policies often used in practice.

Colocation and Placement Optimizations. To improve cluster utilization, existing GPU schedulers often deploy optimizations such as space sharing as in Gandiva [58], where multiple jobs can use the same accelerator concurrently, and placement sensitivity as in Themis and Tiresias [28, 40], which involves the careful placement of tasks in a distributed job to ensure good scaling performance. The performance benefits of these optimizations should be considered explicitly while optimizing for global scheduling objectives, since these optimizations are more effective when deployed in a heterogeneity-aware way. We show that explicit modeling for space sharing can improve objectives by 2.2× compared to Gandiva's ad-hoc approach.

In this paper, we present Gavel, a new cluster scheduler designed for DNN training in both on-premise and cloud deployments, that effectively incorporates heterogeneity in both hardware accelerators and workloads to generalize a wide range of existing scheduling policies. For example, Gavel can provide heterogeneity-aware versions of fair sharing / least attained service [28], FIFO, minimum makespan, minimum cost subject to SLOs, finish-time fairness [40], shortest job first, and hierarchical policies [12, 59].

Gavel's key observation is that many widely used scheduling policies, including hierarchical ones, can be expressed as *optimization problems* whose objective is a function of the jobs' achieved throughputs. For example, least attained service is equivalent to maximizing the minimum scaled throughput among the jobs, makespan is equivalent to minimizing the maximum duration (computed as the ratio of number of iterations to achieved throughput), and so on. Given the optimization problem for a scheduling policy, Gavel introduces a general way to transform the problem to make it heterogenity-, colocation- and placement-aware. In particular, Gavel changes the problem to search over a *heterogeneous allocation* for each job, the fraction of time spent in various resource configurations (e.g., 60% of time running alone on a V100 GPU and 40% of time space-sharing an A100 GPU with another job), and changes the throughput terms in the objective function to *effective throughput*, i.e. the average throughput of the job over the mix of resources in its allocation. Additional constraints need to be added to ensure that the returned allocation is valid. We show that Gavel's transformed optimization problems are efficient to execute even for clusters with hundreds of GPUs and jobs, and can support a wide range of policies. Many of these problems can be solved using a sequence of one or more linear programs.

Gavel's heterogeneity-aware allocations for each job need to be mapped to actual scheduling decisions (placement of jobs on specific resources in the cluster for a specified duration of time). To achieve this, Gavel uses a preemptive *round-based scheduling mechanism* to ensure that jobs receive resources in fractions similar to the computed target allocation. Gavel's scheduling mechanism needs to be able to schedule both distributed training jobs, which request multiple accelerators at once, as well as combinations of jobs running concurrently on a given accelerator due to space sharing.

Gavel makes these scheduling decisions transparently: it specifies an API between the scheduler and applications that allow jobs written in existing deep learning frameworks like PyTorch [48] and TensorFlow [13] to be moved between resources with minimal code changes, and uses a mechanism similar to Quasar [21] to estimate performance measurements of colocated jobs, which are needed as inputs to Gavel's policies, when not available *a priori*.

By explicitly considering performance heterogeneity, Gavel improves various policy objectives (e.g., average job completion time or makespan): on a smaller physical cluster, it improves average JCT by 1.5×, and on a larger simulated cluster, it increases the maximum input load a cluster can support, while improving objectives such as average job completion time by 3.5×, makespan by 2.5×, and cost by 1.4×.

To summarize, our main contributions are:

- A systematic method to convert existing cluster scheduling policies into equivalent policies that consider heterogeneity and colocation; these equivalent optimization problems are practical for current DNN clusters.
- A round-based scheduling mechanism to ensure that the cluster realizes the allocations returned by these policies.
- Generalizations of many existing policies in our framework that improve corresponding objectives.

Gavel is open sourced at https://github.com/stanford-futuredata/gavel.

2 Background

In this section, we provide a brief overview of DNN training (§2.1), and discuss performance optimizations used in existing schedulers that Gavel can help deploy more effectively (§2.2).

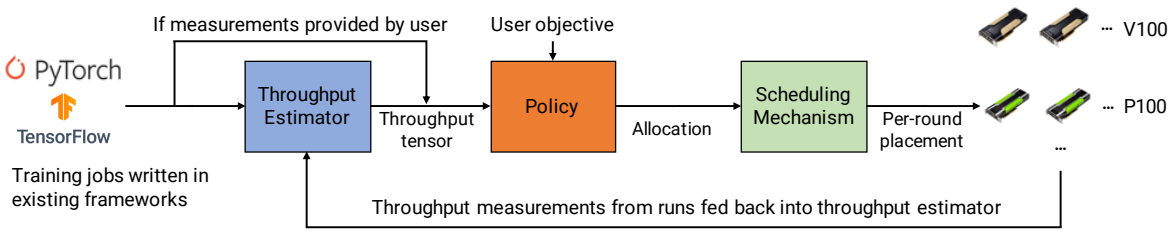


Figure 2: Gavel overview. Jobs are written in frameworks like PyTorch or TensorFlow. Gavel's throughput estimator obtains performance measurements for each runnable job on each available accelerator type if necessary; its policy then computes an allocation that optimizes a user-specified objective such as fairness. Gavel's scheduling mechanism accepts this computed allocation as an input, and makes per-round placement decisions in proportions that faithfully mimic the computed allocation.

2.1 Deep Neural Network (DNN) Training

DNN training proceeds in iterations. In each iteration, the DNN processes a collection of inputs (called a minibatch) and subsequently updates the model parameters using gradients derived from the input minibatch. Each minibatch is typically of similar size, which means model training throughput using short profiling runs (order of minutes). Gavel leverages this fact in its throughput estimator. Jobs are typically fairly long-running (on the order of hours to days), and can be distributed over many workers [9, 58].

Modern DNN schedulers leverage the fact that DNN training is iterative to suspend and resume training at iteration boundaries [28, 58]; this ensures that jobs can be time multiplexed over the existing physical resources. The latest model parameters need to be checkpointed to stable storage when a job is suspended to ensure training progress is not lost. In this work, we show how *time sharing* should be deployed to optimize various single- and multi-job objectives.

2.2 Performance Optimizations

Prior work has shown that GPUs can be severely underutilized in multi-tenant clusters [34]; for example, average GPU utilization (measured as the percentage of GPU Streaming Multiprocessors active over time) was as low as 52% on a Microsoft cluster. Prior work has also shown the placement of tasks for a distributed training job can have significant impact on performance. Gavel can *optionally* deploy these optimizations systematically, as we show in §3.1.

Space Sharing. Smaller models often do not leverage the full computational capacity of modern GPUs. In such cases, concurrently executing multiple models on the same GPU using NVIDIA's Multi Process Service (MPS) or CUDA streams can help improve utilization [10, 47].

Placement Sensitivity. DNN models show heterogeneity in their distributed scaling behavior depending on the size of the tensors that need to be exchanged between workers during training: some models have compact weight representations and can scale well even when workers are not on the same server, while other models scale poorly when workers are spread over many servers. Existing schedulers like Tiresias use heuristics for placement sensitivity.

3 System Overview

Given a collection of jobs, Gavel arbitrates cluster resources (in the form of accelerators of different types) among the resident jobs, while optimizing for the desired cluster objective. This is accomplished in a two-step process: first, a *heterogeneity-aware policy* computes the fraction of time different jobs (and combinations) should run on different accelerator types to optimize the desired objective. These policies require as input the performance behavior (in terms of throughputs) for each job on each accelerator type, which can either be provided by the user, or can be measured on the fly by Gavel's throughput estimator. Allocations are intended to be respected only between allocation recomputation events; for example, if job 1 is much longer than job 2, the allocation will be recomputed once job 2 completes. Gavel can recompute its policy either when a *reset event* occurs (job arrives or completes, worker in the cluster fails), or at periodic intervals of time. Given the policy's output allocation, Gavel's *scheduling mechanism* grants jobs time on the different resources, and moves jobs between workers as necessary to ensure that the true fraction of time each job spends on different resources closely resembles the optimal allocation returned by the policy. Gavel's workflow is shown in Figure 2.

3.1 Heterogeneity-Aware Policies

Gavel expresses scheduling policies as optimization problems for various objectives of interest, such as fairness or makespan, and allocations as matrices that specify the fraction of wall-clock time a job should spend on each accelerator type *between* allocation recomputations. A matrix X can represent allocations on a single accelerator type (homogeneous setting), on multiple accelerator types (heterogeneous setting), as well as with other optimizations. Consider X^{example}:

$$X^{\text{example}} = \begin{array}{c} \begin{matrix} V100 & P100 & K80 \end{matrix} \\ \begin{pmatrix} 0.6 & 0.4 & 0.0 \\ 0.2 & 0.6 & 0.2 \\ 0.2 & 0.0 & 0.8 \end{pmatrix} \end{array} \begin{array}{l} \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{array}$$

According to this allocation specified over three jobs and three accelerator types, job 0 should spend 60% of the time this

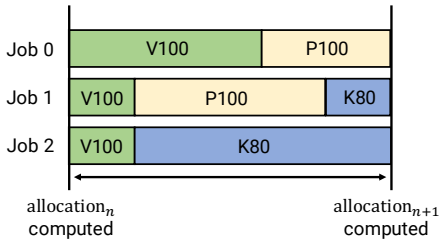


Figure 3: The *cumulative* time each job spends on accelerator types between allocation recomputations for allocation X^{example}.

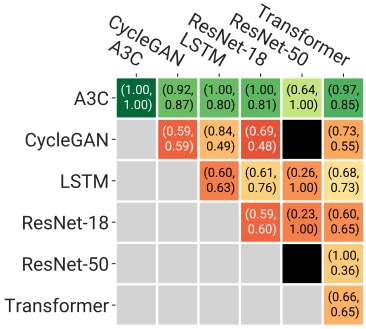


Figure 4: Performance of several DNN models when run concurrently on a single P100 GPU. The cell at row i and column j reports the normalized throughput (iterations/second) achieved by co-located models i and j. Throughputs are normalized with respect to the throughput achieved by each model when run in isolation. Black squares show jobs that cannot co-locate due to memory constraints.

allocation is valid on a V100 GPU, and the remaining 40% of time on a P100 GPU. This is shown visually in Figure 3.

Gavel finds an optimal value for the matrix X given a policy expressed as an optimization problem. To construct the optimization problem for a given policy, Gavel requires a *throughput matrix* T with each job's throughput (in training iterations per second) on different accelerators. T_{mj} can be set to $-\infty$ if job m does not run on accelerator type j (for example, due to memory constraints).

Given T and X, we define the *effective throughput* of a model m as the time-weighted average throughput across accelerators and jobs. We denote this quantity $\text{throughput}_T(m,X)$ or simply $\text{throughput}(m,X)$ (dropping the T) for brevity. For allocations X without space sharing,

$$\text{throughput}(m,X) = \sum_{\substack{j \in \\ \text{accelerator types}}} T_{mj} \cdot X_{mj}$$

Different cluster scheduling policies can be expressed as optimization problems for X while maximizing or minimizing an appropriate objective function. Constraints need to be specified to ensure that X is a valid allocation. A hypothetical policy that maximizes total effective throughput looks like,

$$\text{Maximize}_X \sum_{m \in \text{jobs}} \text{throughput}(m,X)$$

Subject to the following constraints:

$$0 \leq X_{mj} \leq 1 \qquad \forall (m,j) \quad (1)$$
$$\sum_j X_{mj} \leq 1 \qquad \forall m \quad (2)$$
$$\sum_m X_{mj} \cdot \text{scale_factor}_m \leq \text{num_workers}_j \qquad \forall j \quad (3)$$

These constraints ensure that each job-worker allocation is non-negative and between 0 and 1 (equation 1), that the total allocation for a job does not exceed 1 (equation 2), and that the allocation does not oversubscribe workers (equation 3).

Space Sharing. Gavel's allocation matrices can also incorporate space sharing (SS). While previous work has used greedy algorithms for space sharing, we found that different pairs of DNN applications in practice have vastly different performance when colocated together, based on the resources they consume (Figure 4). When using space sharing, X needs to contain rows for each viable combination of jobs, and T needs to have throughputs of the job combinations, like:

$$T = \begin{array}{c} \begin{matrix} V100 & P100 & K80 \end{matrix} \\ \begin{pmatrix} 40.0 & 20.0 & 10.0 \\ 15.0 & 10.0 & 5.0 \\ (20.0, 7.5) & 0.0 & 0.0 \end{pmatrix} \end{array} \begin{array}{l} \\ \text{job 0} \\ \text{job 1} \\ \text{jobs (0, 1)} \end{array}$$

The SS-aware allocation X dictates the fraction of time that each *job combination* should spend on each accelerator type.

We limit entries of T to combinations of at most 2 jobs; we found empirically that larger combinations rarely increase net throughput. Additionally, although the size of T grows quadratically with the number of jobs even with job combinations of size 2, we found that in practice we only need to consider combinations that actually perform well. We evaluate the scaling behavior of these SS-aware policies in §7.4.

Objectives in terms of $\text{throughput}(m,X)$ remain the same; however, $\text{throughput}(m,X)$ now needs to be computed to include the throughputs of co-located jobs:

$$\text{throughput}(m,X) = \sum_{\substack{j \in \\ \text{accelerator types}}} \sum_{k \in C_m} T_{kjm} \cdot X_{kjm}$$

The constraints need to be slighly modified as well to ensure that X is a valid allocation in this new regime:

$$0 \leq X_{kj} \leq 1 \qquad \forall (k,j)$$
$$\sum_{k \in C_m} \sum_j X_{kj} \leq 1 \qquad \forall m$$
$$\sum_k X_{kj} \cdot \text{scale_factor}_m \leq \text{num_workers}_j \qquad \forall j$$

C_m is the set of all job combinations that contain job m.

Placement Sensitivity. Similarly, Gavel's allocation matrices can also be extended to incorporate placement sensitivity. The observed throughput for distributed jobs depends on the location of tasks, as well as the model and accelerator type (slower workers are less likely to be communication-bound,

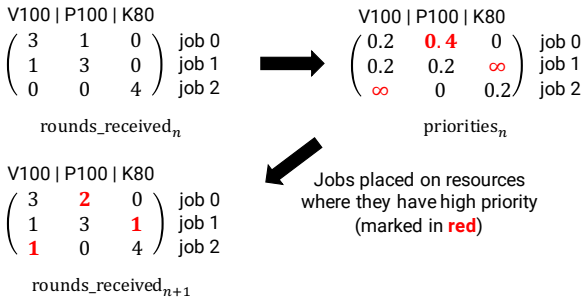


Figure 5: Priorities are used to move the received allocation towards the intended allocation (in this case, X^{example}). priorities$_n$ is computed as X/rounds_received$_n$ (element-wise division).

which means consolidation of tasks is less effective). We can make our policies *placement-sensitive* by considering the performance of distributed jobs in: 1) a consolidated setting, where as many accelerators are on the same server as possible (for example, 8 GPUs per server if using 8-GPU servers), and 2) an unconsolidated setting, where accelerators are on independent servers. These are extreme points in the placement space, and are upper and lower bounds on performance. We can model this in our policies by having two different worker types (consolidated and unconsolidated) with corresponding throughput values in T and allocation values in X.

3.2 Round-based Scheduling Mechanism

After computing the optimal allocation, Gavel's next step is to assign jobs (or job combinations, in the case of SS) to accelerator types while matching the optimal allocation as closely as possible. That is, to realize the allocation X^{example} above, the scheduling mechanism needs to make sure that in the time period where jobs 0, 1, and 2 are the only three runnable jobs in the cluster, jobs should receive resources according to their computed optimal time fractions.

To do this, the scheduler computes a priority score for every job and accelerator type combination that is high when a job has received a smaller time fraction than the optimal allocation. Scheduling is performed in rounds; in each round, the scheduler runs jobs in decreasing priority order, while ensuring that a given job is not scheduled on multiple workers (or accelerators) in a given round. This is shown in Figure 5. Priorities are updated as rounds complete. We have found empirically that round durations of around 6 minutes allow Gavel to effectively approximate the ideal allocation (§7.5).

3.3 Throughput Estimator

To estimate the throughputs of concurrent jobs (e.g., in the case of space sharing), Gavel employs a throughput estimator, similar to those found in prior work such as Quasar [21]. Gavel's throughput estimator maps a new job to a set of pre-profiled reference jobs. The throughputs of the closest reference job can then be used as the initial performance estimate for the new job's combinations. For individual jobs, the throughput estimator is not needed, since throughputs can be estimated on the fly as jobs run on different resource types.

3.4 Limitations and Non-Goals

While Gavel exposes a flexible API that supports a variety of policies and objectives, we do not propose new scheduling policies or performance optimizations in this work. Instead, Gavel's main goal is to determine how best to share resources amongst many different users and jobs in a heterogeneity-aware way, while supporting many existing cluster-wide objectives. Gavel accomplishes these goals with a policy framework that easily allows policies to be made heterogeneity-, colocation-, and placement-aware (§4), a reusable scheduling mechanism (§5), and a narrow scheduler API that allows users to deploy their applications with minimal code changes (§6).

4 Scheduling Policies

In this section, we show how various scheduling policies such as max-min fairness (Least Attained Service or LAS) and multi-level fairness can be expressed as optimization problems in terms of effective throughput. We describe some properties of the resulting heterogeneity-aware allocations at the end of this section.

4.1 Max-Min Fairness as an Optimization Problem

The classical Least Attained Service (LAS) policy, used by Tiresias [28], implements max-min fairness across active users in the cluster, by round-robining resources across jobs according to the total number of accelerator hours consumed. This can be modified into a weighted max-min fairness policy with per-user weights w_m. On a homogeneous cluster, if a job m with weight w_m receives a fraction X_m (which is a scalar since there is only one resource type), LAS can be expressed as the following optimization problem:

$$\text{Maximize}_X \min_m \frac{1}{w_m} X_m$$

We need to add an additional constraint to ensure that the cluster is not overprovisioned ($\sum_m X_m \leq 1$).

However, this vanilla LAS policy is not fair in a heterogeneous setting; jobs might see unequal reductions in throughput due to variations in performance across accelerator types. For example, giving one job a K80 and another job a V100 would equalize their number of resources, but could result in very low performance for the job with the K80.

To compute a more fair allocation, we can compute max-min fairness over the weighted normalized effective throughputs, as defined in §3.1. Let X_m^{equal} be the allocation given to job m assuming it receives equal time share on each worker in the cluster. For example, if the cluster had 1 V100 and 1 K80, $X_m^{\text{equal}} = [0.5, 0.5]$. X_m^{equal} scales the effective throughputs to make them comparable across jobs.

$$\text{Maximize}_X \min_m \frac{1}{w_m} \frac{\text{throughput}(m, X)}{\text{throughput}(m, X_m^{\text{equal}})}$$

Policy	Description
Makespan	Minimize time taken by batch of jobs.
LAS [28]	Max-min fairness by total compute time.
LAS w/ weights	Max-min fairness with weights.
Finish Time Fairness [40]	Maximize minimum job speedup.
FIFO	First in, first out.
Shortest Job First	Minimize time taken by shortest job.
Minimize cost	Minimize total cost in public cloud.
Minimize cost w/ SLOs	Minimize total cost subject to SLOs.
Hierarchical [59]	Multi-level policy: FIFO, fairness, etc.

Table 1: Policies that can be expressed in Gavel.

As specified in §3.1, additional constraints need to be specified to ensure that allocations are valid.

As an example, consider 3 jobs which benefit differently when moved from a K80 GPU to a V100 GPU:

$$T = \begin{array}{c} \begin{matrix} V100 & K80 \end{matrix} \\ \begin{pmatrix} 40.0 & 10.0 \\ 12.0 & 4.0 \\ 100.0 & 50.0 \end{pmatrix} \end{array} \begin{matrix} \\ \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{matrix}$$

Solving the above optimization problem with $w_m = 1$, and a cluster with 1 V100 and 1 K80 yields the following allocation:

$$X^{\text{het.}} = \begin{array}{c} \begin{matrix} V100 & K80 \end{matrix} \\ \begin{pmatrix} 0.45 & 0.0 \\ 0.45 & 0.09 \\ 0.09 & 0.91 \end{pmatrix} \end{array} \begin{matrix} \\ \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{matrix}$$

Jobs receive about 10% higher throughput compared to an allocation where every user is given $1/n$ of the time on each accelerator (here, $n = 3$), also called an *isolated allocation* [26].

Fairness policy objective functions need to be modified to take into account muti-resource jobs with $\text{scale_factor}_m > 1$, since these multi-resource jobs occupy a larger share of the cluster per unit time. An easy way to do this is to multiply the max-min objectives from before by scale_factor_m. Concretely, the LAS objective from before now becomes,

$$\text{Maximize}_X \min_m \frac{1}{w_m} \frac{\text{throughput}(m,X)}{\text{throughput}(m,X_m^{\text{equal}})} \cdot \text{scale_factor}_m$$

4.2 Other Policies as Optimization Problems

We can express many other common cluster scheduling policies, some proposed by recent papers, using $\text{throughput}(m,X)$; we list these policies in Table 1. Most of these policies can be expressed using a single linear program, with a few exceptions: the cost policies are formulated as a linear-fractional program [8], which can be reduced to a sequence of linear programs. These optimization problems yield corresponding heterogeneity-aware allocations. The optimal allocation can be computed using off-the-shelf solvers.

Minimize Makespan. The makespan minimization policy tries to complete all active jobs as soon as possible. Gandiva uses a version of this policy to finish higher-level tasks such as hyperparameter tuning and AutoML, which involve training a large number of variants of a model. If num_steps_m is the number of iterations remaining to train model m, then the makespan is the maximum of the durations of all active jobs, where the duration of job m is the ratio of the number of iterations to $\text{throughput}(m,X)$ (expressed in iterations / second). Overall, this can be framed as,

$$\text{Minimize}_X \max_m \frac{\text{num_steps}_m}{\text{throughput}(m,X)}$$

Minimize Finish-Time Fairness (Themis). Themis [40] proposes a new metric called finish-time fairness (represented as ρ), which is the ratio of the time taken to finish a job using a given allocation and the time taken to finish the job using $1/n$ of the cluster (X^{isolated}), assuming n users using the cluster. This can be expressed in terms of $\text{throughput}(m,X)$ as follows (num_steps_m is the number of iterations remaining to train model m, t_m is the time elapsed since the start of training for model m, and t_m^{isolated} is the hypothetical time elapsed since the start of training if model m had $1/n$ of the cluster to itself),

$$\rho_T(m,X) = \frac{t_m + \frac{\text{num_steps}_m}{\text{throughput}(m,X)}}{t_m^{\text{isolated}} + \frac{\text{num_steps}_m}{\text{throughput}(m,X^{\text{isolated}})}}$$

The final optimization problem is then,

$$\text{Minimize}_X \max_m \rho_T(m,X)$$

FIFO. The First-In-First-Out (FIFO) policy schedules jobs in the order they arrive. In a heterogeneous regime, jobs should be placed on the fastest available accelerator type. Mathematically, we can write this as maximizing the throughput of job m relative to its throughput on the fastest type ($\text{throughput}(m,X^{\text{fastest}})$). Assuming that jobs are enumerated in order of their arrival time (m arrived before $m+1$), a FIFO allocation can be computed with the following objective:

$$\text{Maximize}_X \sum_m \frac{\text{throughput}(m,X)}{\text{throughput}(m,X^{\text{fastest}})}(M-m)$$

where M is the total number of jobs.

Shortest Job First. The Shortest Job First policy finds the allocation that minimizes the duration of the shortest job,

$$\text{Minimize}_X \min_m \frac{\text{num_steps}_m}{\text{throughput}(m,X)}$$

Minimizing Total Cost and Cost subject to SLOs. We can express policies for deployments that use elastic public cloud resources. Since cloud VMs are charged on a per-time basis, we can express policies that explicitly optimize for total cost, speed, or both.

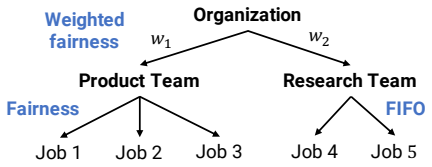


Figure 6: Example of a hierarchical policy: weighted fairness across two entities: a product and research team, fairness across jobs within the product team, and FIFO within the research team.

Consider a simple policy that maximizes total throughput,

$$\text{Minimize}_X \sum_m \text{throughput}(m, X)$$

The above policy can be extended to incorporate cost by optimizing the following cost-adjusted objective,

$$\text{Maximize}_X \frac{\sum_m \text{throughput}(m, X)}{\sum_m (\sum_j \text{cost}_j \cdot X_{mj})}$$

where cost_j is the cost of accelerator type j. The numerator in the above objective is the time-averaged effective throughput, and the denominator is the time-averaged cost. When using space sharing, care must be taken to not double count the cost of instances running job combinations (all jobs in a job combination derive value in the form of some throughput).

Jobs can have time SLOs as well, e.g., certain high-priority jobs might need to complete every 12 hours. We can add additional constraints: given SLO_m for each model m (models without SLOs can have $\text{SLO}_m = \infty$),

$$\text{throughput}(m, X) \geq \text{num_steps}_m / \text{SLO}_m$$

4.3 Hierarchical Scheduling Policies

Modern cluster schedulers do not only deploy "single-level" policies. Hierarchical policies are common [6, 12, 59]: a large organization might share a single physical cluster among many sub-organizations (or entities) using a fairness policy. In turn, each entity can share resources among individual jobs according to a distinct per-entity policy, such as per-user fairness or FIFO. We give an example in Figure 6, where a research and product team share the same physical cluster. The research team runs ad-hoc experiments that can be executed in FIFO order, but the product team needs to ensure that all its jobs receive a fair share of the cluster.

Gavel can currently support fairness in the upper levels and fairness or FIFO in the lower levels, which matches the hierarchical policies supported by the Hadoop scheduler [6]. Determining how to extend this to other hierarchical policy sets (for example, with finish time fairness) is future work.

Gavel solves hierarchical objectives using a procedure called water filling [15], which is used in other max-min fairness problems such as link allocation in networks [49]. At a high level, the water-filling algorithm increases the allocation given to all parties at an equal rate to respect max-min fairness, until a party saturates. The saturated party is then taken out, and the procedure repeated iteratively until all commodities are saturated. We adapt this procedure to our setting, solving a series of optimization problems iteratively: an LP that computes a fair allocation across entities while respecting each entity's internal policy, and an MILP that identifies *bottlenecked jobs*, i.e., jobs whose effective throughputs cannot be improved without lowering other jobs' effective throughput.

We assume that each entity s is associated with a weight w_s; the jobs belonging to this entity receive a total cluster share proportional to this weight. We denote w_m^{job} to be the weight of job m, set such that $\sum_{m \in s} w_m^{\text{job}} = w_s$. Jobs are assigned priorities in accordance to the relevant entity's policy; for example, a fairness policy within an entity would assign each job a weight proportional to its individual weight within the entity, while for FIFO, the first job in the queue would initially receive the entire weight of the entity.

In each iteration, we solve the following modified LP (assuming $\text{scale_factor}_m = 1$ for all m for simplicity):

$$\text{Maximize}_X \min_{\{m : w_m^{\text{job}} > 0\}} \frac{1}{w_m^{\text{job}}} \left(\frac{\text{throughput}(m, X)}{\text{throughput}(m, X_m^{\text{equal}})} - t_m \right)$$

t_m is the normalized effective throughput of job m in the previous iteration ($t_m := 0$ in the first iteration). The above objective can be appropriately modified for $\text{scale_factor}_m > 1$. Bottlenecked jobs are given priority 0 and no longer considered in future iterations. Priorities are redistributed among non-bottlenecked jobs according to the entity's policy at the end of every iteration. For instance, in the example shown in Figure 6, if job 4 is bottlenecked, then its weight is reassigned to job 5 in accordance to the FIFO policy, while if job 2 is bottlenecked, its weight is distributed equally between jobs 1 and 3 in accordance with the entity's fairness policy. The LP then solves the max-min problem on the resources remaining while ensuring each job's throughput does not drop compared to the previous iteration's allocation X^{prev}, expressed as $\text{throughput}(m, X) \geq \text{throughput}(m, X^{\text{prev}})$ for all m. Iterations continue until all jobs are bottlenecked. To make this procedure more concrete, consider an example with 4 identical jobs: job 1 with a weight of 3.0, and jobs 2 to 4 with a weight of 1.0; and 4 identical GPUs. In the first iteration, job 1 is assigned resources such that its throughput is 1.0, and jobs 2, 3, and 4 are assigned resources such that their throughput is 0.33 to respect weights. Job 1 is a bottleneck; the throughput of the remaining jobs can still be increased. In the next iteration, jobs 2 to 4 are given full-GPU allocations.

The final allocation satisfies both inter-entity and intra-entity policies. We note that the above water-filling procedure can also be used for single-level fairness policies such as the one described in §4.1 to improve the throughput of non-bottelenecked jobs.

4.4 Properties of Gavel's Policies

Existing scheduling schemes have been analyzed in terms of properties like sharing incentive, Pareto efficiency, and strategy proofness [26]. We formalize Gavel's heterogeneity-aware policies in the context of these properties as well.

Homogeneous Clusters. For homogeneous clusters, Gavel's heterogeneity-aware policies are equivalent to the baseline policies (throughput$(m,X) = X_m \cdot T_m$), since the heterogeneity-aware optimization problems reduce to the original optimization problems with one accelerator type.

Sharing Incentive. For heterogeneous clusters, the policy's objective metric (maximize least job share in LAS, completion time of first job in FIFO, or makespan) is at least as well off as it would be under a policy that naïvely splits all resources equally among all runnable jobs. This is because the allocation corresponding to giving each user $1/n$ of each resource is a feasible solution to Gavel's optimization problem, so Gavel's solution will be at least as good. All Gavel policies have *sharing incentive* [26], which encourages users to use the shared cluster rather than a static private share.

Colocation. Solutions with colocation are always at least as good as without colocation.

Pareto Efficiency. Allocations of max-min fairness policies with water filling are Pareto efficient: that is, the allocation for a particular job cannot be increased without decreasing the allocation for another job.

Note that some of Gavel's policies may not satisfy other desirable properties. For example, Sun et al. [53] showed that no fair-sharing policy can simultaneously satisfy Pareto efficiency, sharing incentive and strategy proofness in a setting with interchangeable resources. If users manipulate their throughputs, then they can possibly obtain larger shares of the cluster (e.g., jobs can be placed on a faster accelerator type) for certain objectives. Exploring how to make Gavel's policies strategy-proof is interesting future work.

5 Scheduling Mechanism

Gavel's scheduling mechanism schedules training iterations of runnable jobs on the available workers (with possibly different accelerators), such that for each schedulable job (or combination), the fraction of wall-clock time it spends on each accelerator type is approximately equal to the computed optimal allocation X^{opt} *between* allocation recomputation events. This is challenging for two main reasons: 1) Jobs can run on multiple accelerators. Moreover, since distributed training can be communication intensive [19, 46], jobs should be placed on accelerators "close" to each other (for example, on accelerators on the same server, or on accelerators in servers in the same rack). 2) Combinations of up to two jobs can run on a set of accelerators in order to improve resource utilization (space sharing). Each distinct job can have ≤ 1 job combination running in a given round to prevent work duplication.

Gavel makes its scheduling decisions in *rounds*. This is similar in spirit to Tiresias's [28] priority discretization in some respects. However, Gavel's scheduling mechanism differs from Tiresias's in three ways:

- Gavel needs to schedule jobs on different accelerator types: it needs to decide which job should be active in any round *and* which accelerator type to use.
- Gavel needs to grant resources to jobs while respecting an *arbitrary allocation* returned by the policy.
- Gavel's round-based scheduler grants time to jobs while ensuring that multiple job combinations sharing a job do not run in the same round; Tiresias does not consider job combinations and does not need to deal with this.

Gavel's scheduler tries to place work on all available workers for a specific duration (this time period is configurable; we use 6 minutes in our experiments). We call the work handed to each worker in a given round a *micro-task*. Without rounds, jobs that request many accelerators can suffer from starvation. For example, consider a cluster with 8 total accelerators and 4 available. The scheduler can handle a 8-accelerator job waiting for resources in one of two ways: a) wait for 8 accelerators to become available; 4 accelerators will be unused until the full quota of 8 accelerators becomes available, b) keep the 8-accelerator job in the queue, and give 4 accelerators to another job that requests a fewer number of resources. However, this situation can repeat itself, leading to starvation [59]. Scheduling is thus performed in rounds to limit resource under-utilization, simplify scheduling logic, and ensure that jobs with large scale factors do not experience prolonged starvation.

Since the number of active, *schedulable* jobs might far exceed the total number of workers, Gavel first determines the job combinations that should run in the upcoming round. To do this, Gavel maintains the time t_{mj} spent by a job (or combination) m on accelerator type j, which is updated as jobs run on different accelerator types every round. Given t_{mj}, Gavel's scheduler can then compute the fraction of total wall-clock time spent by each job (or combination) m on each accelerator type j as $f_{mj} = t_{mj}/(\sum_{m'} t_{m'j})$. The matrix of priorities is then just the element-wise division of X^{opt} by f.

Algorithm. In every round, we want to move f_{mj} closer to X^{opt}_{mj}. This can be achieved by giving high-priority jobs time on accelerator type j.

This problem can be solved exactly if jobs only request single accelerators and if space sharing is not deployed by finding the num_workers$_j$ jobs with highest priority (for example, using a heap). However, jobs submitted to Gavel can be distributed, and space sharing can be used to improve resource utilization. Solving this problem exactly with these added requirements makes the problem similar to a multiple-choice knapsack problem [52], which is NP-hard.

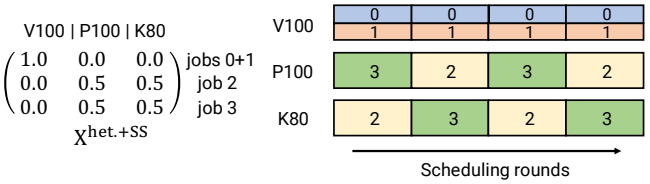


Figure 7: Round-based scheduling mechanism in action to achieve an allocation $X^{\text{het.+SS}}$. Space sharing is shown with vertically split boxes. Each round is denoted by a box.

Algorithm 1 Algorithm for Gavel's scheduling mechanism

```
function SCHEDULE_JOBS
    active_combinations ← all active job combinations
    num_workers_rem. ← number of total workers
    while num_workers_rem.g > 0 do
        j ← job combination with highest priority
        Remove j from active_combinations
        if j.scale_factor > num_workers_rem. then
            continue
        for all j' that conflict (share a job k) with j do
            Remove j' from active_combinations
        num_workers_rem. −= j.scale_factor
```

To overcome these challenges, we observe that it is acceptable to make greedy sub-optimal *scheduling* decisions occasionally in any given round, since we can recover from these sub-optimal decisions in subsequent rounds: our goal is to ensure that the average allocation each job receives *over multiple rounds* resemble the computed allocation (the allocations returned by policies are optimal, which follows from how policies in Gavel are expressed as optimization problems). We study the impact of this design choice in §7.5. A job (combination) not run in a particular round will have increased priority in subsequent rounds until it receives accelerator time, while a job that runs in a particular round will have decreased priority. This ensures that jobs do not suffer from starvation if they have a non-zero optimal allocation.

Gavel uses a greedy algorithm to pick the highest-priority job combinations that fit in the provided resource budget. The algorithm maintains a set of eligible job combinations (`eligible_job_combinations`) that can be scheduled in the upcoming scheduling round. The scheduling mechanism then tries to add job combinations with highest priority into a `job_combinations_to_schedule` set. Once a job combination is added to this set, all *conflicting* job combinations are removed from the set of eligible combinations to ensure that a given job is not run more than once in a given scheduling round. Job combinations that cannot fit in the current round due to space limitations (required number of accelerators unavailable) are also removed from the set of eligible combinations. This procedure is detailed in Algorithm 1. Gavel's scheduling mechanism is decoupled from its policies, ensuring that the same scheduling mechanism can be used for many different policies. Figure 7 shows Gavel's scheduling mechanism in action.

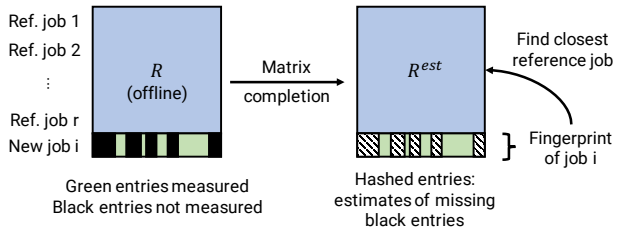


Figure 8: Gavel's throughput estimator. Profiling is combined with matrix completion to obtain a fingerprint for every new job. The fingerprint is then used to find the closest reference job.

Once Gavel has decided what jobs (and combinations) should run in a given round on different accelerator types, Gavel must decide how to *place* these jobs. Gavel's scheduler places jobs in decreasing order of the number of requested workers, and tries to give jobs accelerators on the same physical server to minimize fragmentation.

6 Implementation

We implemented a prototype of Gavel in approximately 9000 lines of Python code, and implemented a simulator in about 500 LOC. We used `cvxpy` [23] to implement Gavel's heterogeneity-aware policies, and `gRPC` [4] to communicate control messages between the scheduler and workers.

Interface between Scheduler and Applications. Gavel currently supports user applications written in PyTorch [48]; support for TensorFlow [13] is left for future work. The scheduler and user applications then interact through a narrow API. Gavel ships with a Python library that users can import into their code. This library provides an implementation for a wrapper around existing framework-provided data iterators (`GavelIterator`). `GavelIterator` ensures that each task in a distributed job runs for the same number of iterations, and synchronizes the conclusion of rounds between the scheduler and workers. `GavelIterator` is instantiated with arguments `train_loader` (base data loader), `load_checkpoint`, `save_checkpoint`, and a configuration object. `load_checkpoint` is a pointer to a function that loads all necessary parameters and metadata from a checkpoint at the start of a round, and `save_checkpoint` is a pointer to a function that creates a checkpoint at the end of a round; these need to call appropriate framework methods ($<$ 5 LOC).

`GavelIterator` contacts the scheduler near a round end to see if the same job will run in the next round on the same worker. We call this a *lease renewal*. If the lease is not renewed, the iterator calls `save_checkpoint` at round end. The scheduler can then launch another job on the worker.

Throughput Estimation. Gavel uses a similar technique to Quasar [21] to estimate colocated throughputs when using the optional space sharing optimization (if they are not available a priori), mixing profiling with matrix completion.

Model	Task	Dataset / Application	Batch size(s)
ResNet-50 [5, 31]	Image Classification	ImageNet [22]	16, 32, 64, 128
ResNet-18 [31, 39]	Image Classification	CIFAR-10 [36]	16, 32, 64, 128, 256
A3C [27, 44]	Deep RL	Pong	4
LSTM [11]	Language Modeling	Wikitext-2 [42]	5, 10, 20, 40, 80
Transformer [33, 55]	Language Translation	Multi30k [24] (de-en)	16, 32, 64, 128, 256
CycleGAN [38, 60]	Image-to-Image Translation	monet2photo [60]	1
Recoder [45] (Autoencoder)	Recommendation	ML-20M [29]	512, 1024, 2048, 4096, 8192

Table 2: Models used in the evaluation.

Trace	System	Objective	Physical	Simulation
Continuous	Gavel	Average JCT	3.4 hrs	3.7 hrs
Continuous	LAS	Average JCT	5.1 hrs	5.4 hrs
Static	Gavel	Makespan	17.7 hrs	17.6 hrs
Static	Gandiva	Makespan	21.3 hrs	22.1 hrs

Table 3: Comparison of end objective between physical experiment and simulation for two different traces. For the continuous trace, we measure the average JCT of 25 jobs in a steady-state cluster. For the static trace, we measure the total time needed to complete 100 jobs submitted at the start of the run. The heterogeneity-aware policies improve target objectives, and results on the physical cluster are in agreement with results on simulated cluster ($< 8\%$).

Model	Overhead without lease renewals	Overhead with lease renewals
ResNet-18	0.94%	0.17%
ResNet-50	1.58%	0.25%
A3C	0.22%	0%
LSTM	2.91%	0.47%
Transformer	0.77%	0.11%
CycleGAN	0.77%	0.11%

Table 4: Overhead of using preemptive scheduling in Gavel, with and without lease renewals, and with a round duration of 6 minutes.

Matrix completion enables sparse low rank matrices to be reconstructed with low error [17, 43]. With matrix completion, Gavel is able to extrapolate measurements obtained through direct profiling on separate workers dedicated to profiling, and determine the job's most similar pre-profiled reference job. The throughput estimator can then use the reference job's throughput measurements as an initial throughput estimate. Gavel's throughput estimator is diagrammed in Figure 8.

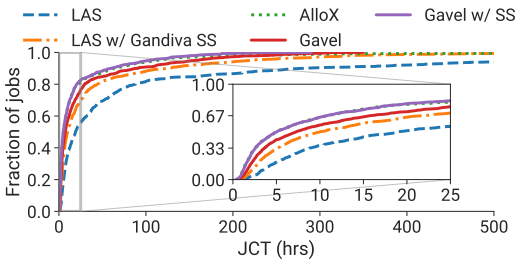
7 Evaluation

In this section, we seek to answer the following questions:

- Do Gavel's heterogeneity-aware policies improve objective metrics in a physical cluster (§7.2) and in simulations of larger clusters (§7.3)?
- How do Gavel's policies scale? (§7.4)



(a) Average job completion time vs. cluster load.



(b) CDF of job completion times (input job rate = 5.6 jobs/hr).

Figure 9: Comparison of heterogeneity-agnostic least attained service (LAS) policy to a heterogeneity-aware LAS policy (Gavel), in simulation on the continuous-single trace.

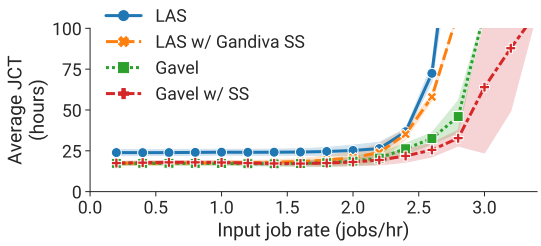
- How well does Gavel's scheduling mechanism realize Gavel's heterogeneity-aware allocations? (§7.5)
- Is Gavel able to accurately estimate the throughputs of co-located jobs when using space sharing? (§7.6)

7.1 Experiment Setup

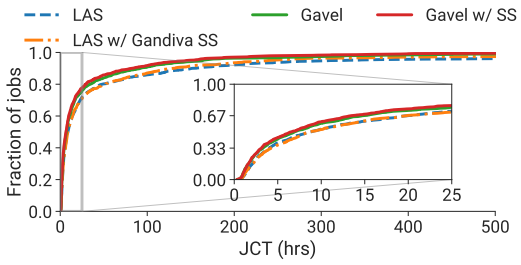
We run experiments on both a physical and simulated cluster.

Clusters. We run physical cluster experiments on a cluster with 8 V100s, 16 P100s, and 24 K80s. Simulated cluster experiments are run on a cluster with 36 GPUs of each type.

Traces. We run physical and simulated experiments on two types of traces: one where all jobs are available at the start of the trace and jobs are *not* subsequently added ("static"), and another where jobs are continuously added to the cluster ("continuous"). For the continuous trace, job arrival times are generated according to a Poisson arrival process with an inter-arrival rate λ. For the simulated experiments, we vary λ to show the extra load each heterogeneity-aware policy is able to sustain in steady state. We run 3 seeds for every λ, and show standard deviations. For the physical cluster experiments, we use a single λ that keeps the cluster well-utilized in steady state. The online traces used in the simulated experiments have a variable number of jobs (at least 5000) and span 20-30 days. We measure the completion times of jobs with ID 4000 to 5000 to study steady state behavior (new jobs continue to be added until jobs of interest complete). Job types are uniformly sampled from the job table with 26 distinct job (or model) types, shown in Table 2. The online traces used in the physical experiments span a day and have 100 jobs.



(a) Average job completion time vs. cluster load.



(b) CDF of job completion times (input job rate = 2.6 jobs/hr).

Figure 10: Comparison of heterogeneity-agnostic least attained service (LAS) policy to a heterogeneity-aware LAS policy (Gavel), in simulation on the continuous-multiple trace. Each input job rate is run with 3 seeds; shaded regions show the standard deviation.

The duration of each job *on a V100 GPU* is sampled from an exponential distribution: jobs have duration 10^x minutes, where x is drawn uniformly from $[1.5, 3]$ with 80% probability, and from $[3, 4]$ with 20% probability. Given the job's observed throughput on the V100 GPU, the number of training steps is then inferred by multiplying the throughput (in steps/sec) by the duration. This matches the process used by Gandiva [58]. For the simulated experiments, we show results in two regimes: one where all jobs use a single worker ("continuous-single"), and another where 70% of jobs request a single worker, another 25% request between 2 and 4 workers, and the remaining 5% request 8 workers, as observed in published traces from Microsoft [9] ("continuous-multiple").

Metrics. For fairness and FIFO policies, our target metric is average job completion time of steady-state jobs, which is the same metric used by related work [28, 41]. We also show finish time fairness (FTF) for policies that explicitly optimize for FTF. For makespan policies, our target metric is the time needed to complete a job batch. For cost-related policies, the metric is cost (in dollars), and the percentage of jobs that violate time SLOs.

7.2 End-to-End Results on Physical Cluster

For our physical cluster experiments, we run a heterogeneity-aware and a heterogeneity-agnostic fairness policy on a continuous trace, and a heterogeneity-aware makespan policy against a baseline that uses Gandiva's ad-hoc space sharing on a static trace. Results are shown in Table 3. Gavel's heterogeneity-aware policies improved average job completion time by **1.5**× and makespan by **1.2**×. For the makespan objective, we do not run Gavel with space sharing; in theory, space sharing would additionally reduce makespan.

We also compare the real performance to simulations and observe that for both policies, the difference between metrics in simulation and on the physical cluster is small ($< 8\%$), indicating that our simulator has high fidelity.

Table 4 shows the overhead of using Gavel's preemptive scheduler with a round duration of 6 minutes, with and without lease renewals. Allocations and worker assignments can be computed asynchronously. The only synchronous overhead is the loading and saving of checkpoints, which is dependent on the size of the model. Lease renewals decrease this overhead by allowing jobs to run on the same worker for extra rounds. The overhead of preemption, even without lease renewals and with a short round duration, is low ($< 3\%$).

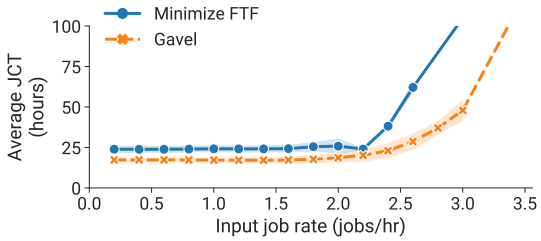
7.3 End-to-End Results in Simulation

We use a larger simulated cluster to evaluate the efficacy of Gavel's heterogeneity-aware policies across a range of objectives, and compare with heterogeneity-agnostic versions from previous work using a round duration of 6 minutes. As appropriate, we compare to other baselines like AlloX. Magnitudes of speedups are higher for these experiments compared to the physical cluster experiments since the simulated traces show job behavior over weeks, while the physical cluster traces are only a day long; consequently, queue buildups are less extreme for the traces used in the physical cluster experiments.

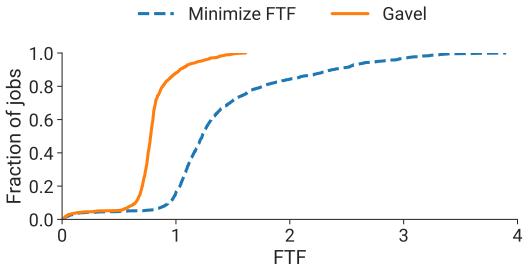
Least Attained Service (LAS). Figures 9 and 10 compare the vanilla LAS policy with its heterogeneity-aware variants. We compare with two other baselines: a modified LAS policy that uses Gandiva's ad-hoc space sharing, and an AlloX policy that explicitly optimizes average job completion time (but only for single-worker jobs). We make three observations.

First, the heterogeneity-aware policies support higher load on the *same* cluster, reduce average JCT by **3.5**× for the continuous-single trace, and by **2.2**× for the continuous-multiple trace (graph can be read by comparing average JCT value for a given input job rate or x-intercept) at high load (5.6 jobs/hr for continuous-single, 2.6 jobs/hr for continuous-multiple). Second, the heterogeneity-aware LAS policy supports higher load than AlloX, since AlloX can give short jobs preferential treatment in the interest of optimizing average JCT, leading to long jobs experiencing starvation (long tail in JCT CDF). At moderate load, AlloX represents a best-case scenario since it explicitly optimizes for average JCT on a heterogeneous cluster. Gavel is able to essentially match this best case scenario, while also supporting other objectives. Third, Gandiva-style packing, which randomly explores job combinations until a combination that improves performance is found, is ineffective compared to Gavel's principled packing (**2.2**× better average JCT for both traces at high load).

Finish Time Fairness (FTF). We compare the heterogeneity-aware version of Finish Time Fairness



(a) Average job completion time vs. cluster load.



(b) CDF of finish time fairness metric (input job rate = 2.6 jobs/hr).

Figure 11: Comparison of a heterogeneity-agnostic policy that optimizes for finish time fairness ("Minimize FTF") to a heterogeneity-aware one (Gavel), in simulation with the continuous-multiple trace.

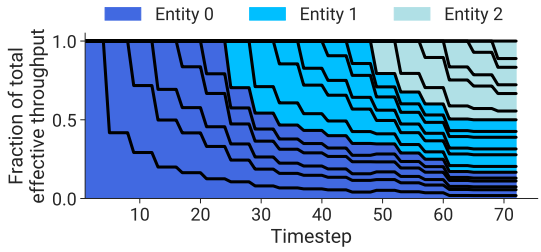
(FTF) to its heterogeneity-agnostic counterpart in Figure 11. The heterogeneity-aware policy reduces average JCTs by **3**× and improves average FTF by **2.8**×. FTF is the ratio of the time taken to finish a job using a given allocation and the time taken to finish the job using $1/n$ of the cluster (X^{isolated}), assuming n users use the cluster. Lower FTF means jobs take less time with the provided allocation compared to X^{isolated}.

Makespan. Gavel's heterogeneity-aware makespan policy reduces makespan by **2.5**× compared to a FIFO baseline, and by **1.4**× compared to a baseline that uses Gandiva's ad-hoc space sharing. Makespan is reduced by a further **8%** when the number of jobs in the trace is high when using space sharing.

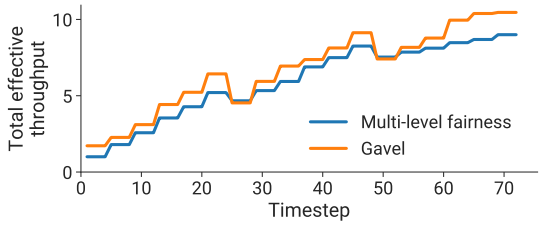
FIFO. The heterogeneity-aware versions of FIFO allow the cluster to support average input job rate. At high load, the heterogeneity-aware version without space sharing reduces average JCT by **2.7**×, and the heterogeneity-aware version with space sharing reduces average JCT by **3.8**× at high load. Space sharing is less effective for distributed jobs: it reduces average JCT by **1.1**× with distributed jobs, compared to **1.4**× for the continuous-single trace.

LAS with priorities. We also run an experiment with the LAS policies where 20% of jobs have higher priority. At high load, Gavel reduces the average JCT of high-priority jobs by **1.5**× and the average JCT of low-priority jobs by **2.7**×.

Cost. We simulate each of the cost policies on a 500-job workload comprised of ResNet-50 and A3C jobs. As we observe in Figure 1b, the ResNet-50 job has the best cost-normalized throughput on the V100 while the A3C job has



(a) Fraction of total throughput for each job with time.



(b) Total throughput vs. time.

Figure 12: Behavior of a multi-level fairness policy with time as jobs are added to a small cluster with 3 V100 GPUs, 3 P100 GPUs, and 3 K80 GPUs. Each line represents a separate job, and jobs are added every 4 timesteps. The first 6 jobs belong to entity 0 (weight of entity, $w_0 = 1$), the next 6 jobs belong to entity 1 ($w_1 = 2$), and the last 6 jobs belong to entity 2 ($w_2 = 3$).

the best cost-normalized throughput on the K80. Each job's duration is chosen from $\{0.5, 1, 2, 4, 8\}$ days, and each job's SLO is chosen from $\{1.2\times, 2\times, 10\times\}$ its duration.

The policy that minimizes cost reduces the total cost compared to the policy that maximizes throughput by a factor of roughly **1.4**×. However, approximately **35%** of jobs violate their SLO as this policy prioritizes cheaper but slower GPUs; in particular, the A3C jobs are scheduled on K80 GPUs which results in violations for tight SLOs. In comparison, the policy that includes SLOs as well eliminates all violations for a small increase in cost (a cost reduction of **1.2**× compared to the baseline policy), by ensuring that A3C jobs with tight SLOs are run on instances with V100 GPUs.

Multi-level Hierarchical Policies. Figure 12 shows the behavior of a multi-level fairness policy as new jobs belonging to multiple entities are added to a heterogeneous cluster with equal numbers of K80, P100, and V100 GPUs. Resources are granted to jobs in a way that respects both the higher-level and lower-level policies: in Figure 12a, fairness is enforced both within and across entities (as can be seen by the widths of the colored bands, which represents cross-entity fairness, and the widths of bands within a color, which represents fairness across jobs within an entity), and allocations are adjusted as new jobs come in. Figure 13 shows results with a fairness+FIFO policy; later jobs in each entity 0 do not receive any GPU time to respect the per-entity FIFO policy.

The multi-level fairness policy can also be implemented in a heterogeneity-agnostic manner by statically partitioning

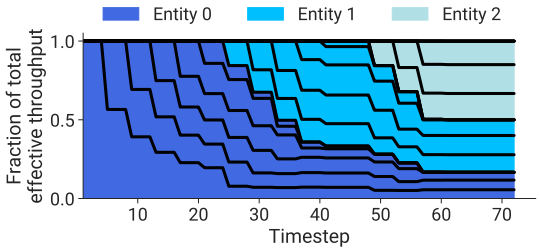
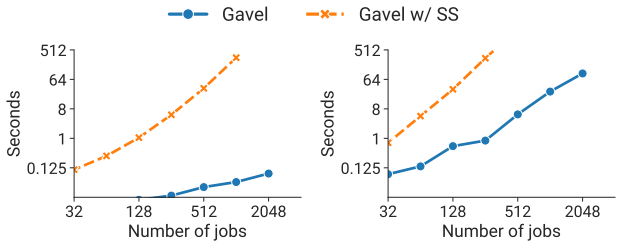


Figure 13: Behavior of a hierarchical policy (weighted fairness as top-level policy, FIFO as bottom-level policy) with time as jobs are added to a small cluster with 3 V100 GPUs, 3 P100 GPUs, and 3 K80 GPUs. Each line represents a separate job, and jobs are added every 4 timesteps. The first 6 jobs belong to entity 0 (weight of entity, $w_0 = 1$), the next 6 jobs belong to entity 1 ($w_1 = 2$), and the last 6 jobs belong to entity 2 ($w_2 = 3$).



(a) LAS. **(b)** Hierarchical.

Figure 14: Scaling of LAS and hierarchical policies with the number of active jobs on a heterogeneous cluster with an equal number of V100, P100, and K80 GPUs. The size of the cluster is increased as the number of active jobs is increased.

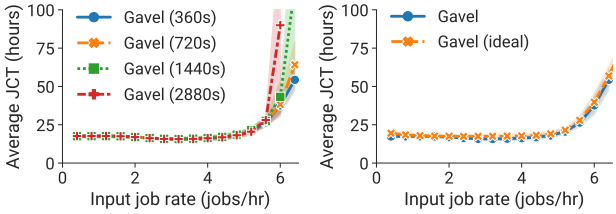
resources across users while respecting per-entity and per-user weights. While this results in a fair allocation as well, we observe that total effective throughput is about **17%** lower compared to the heterogeneity-aware policy (Figure 12b).

7.4 Scalability of Heterogeneity-Aware Policies

Figure 14 shows the scaling behavior of the heterogeneity-aware LAS and multi-level fairness policies with and without space sharing. We observe that even with 2048 active jobs, the hierarchical policy without space sharing can be run in < 10 minutes. With space sharing, the policy can be run with 512 jobs in < 10 minutes. The single-level LAS policy is much cheaper to compute in comparison. We note that allocations do not need to be recomputed every scheduling round – however, the longer the policy takes to run, the longer it takes for the new allocation to be acted upon (jobs can still be given heterogeneity-agnostic allocations in the interim, and consequently time on resources). We believe latencies of < 30 minutes for large clusters are still preferable to non-preemptive schedulers where jobs experience large queuing delays, or preemptive schedulers with heterogeneity-agnostic policies which lead to worse objective values, as shown above.

7.5 Efficacy of Scheduling Mechanism

Figure 15a shows the effect of the round length on average JCT for the heterogeneity-aware LAS policy with a single-GPU trace. We observed similar behavior on traces with multi-GPU jobs, as well as other policies. A smaller round length gives Gavel's scheduling mechanism more rounds to course correct, allowing the true allocation and computed optimal allocation to more closely match. We found that the time needed to load and save checkpoints for our target models is < 5 seconds, which means that a round length of 6 minutes gives a good tradeoff between fidelity with the optimal allocation and preemption overhead (preemption overhead with 6-minute rounds shown in Table 4).



(a) Effect of round length. **(b)** Mechanism vs. ideal.

Figure 15: (a) Effect of round length on average JCT for the heterogeneity-aware LAS policy. (b) Comparison of scheduling mechanism to an ideal baseline that allocates resources to jobs *exactly* according to the computed allocation for the same policy.



Figure 16: Comparison of SS-aware LAS policy with estimated throughputs, compared to the SS-aware with oracle throughputs and LAS without space sharing on a heterogeneous 12-GPU cluster.

We compare this to an ideal baseline that allocates resources to jobs *exactly* according to their computed allocation. As shown in Figure 15b, Gavel's scheduling mechanism with a round duration of 6 minutes behaves almost identically to this ideal baseline with a single-GPU trace (behavior with a multi-GPU trace is similar). We note that the ideal baseline is impractical to use in practice, since jobs with different scale factors can complete at different times (leading to starvation), and preemptions can be often since allocations for some (job, accelerator type) pairs are small, leading to high overhead.

7.6 Impact of Throughput Estimation

Figure 16 shows the effect of Gavel's throughput estimator on average JCT when using the space sharing-aware LAS policy compared to the LAS policy without space sharing, and the LAS policy with space sharing and oracle throughputs. The throughput estimator is able to determine missing throughputs in an online fashion accurately enough to observe a very small decrease in average JCT at high load (orange and blue lines).

8 Related Work and Discussion

In this section, we compare Gavel to related work.

Existing DNN Training Schedulers. Several recent papers have proposed schedulers targeting DNN training workloads.

Gandiva [58] uses time and space sharing to reduce queuing delay and improve resource utilization, but does not specify an explicit scheduling policy and does not support configurable objectives. It uses a profiling-based methodology to determine whether to co-locate jobs on an accelerator. However, it does not incorporate model performance data (isolated or co-located performance) explicitly into its scheduling policy, resorting to random exploration of job combinations until a combination that improves performance is found.

Tiresias [28] and Themis [40] use different objectives to achieve multi-job fairness. However, both do not incorporate jobs' affinities for different accelerator types in their scheduling objectives, and have scheduling mechanisms strongly coupled with the target policy, making it hard to support other more sophisticated policies like multi-level fairness.

AlloX [37] and Gandiva$_{\text{fair}}$ [18] are recent DNN schedulers that do consider worker and model heterogeneity. However, both only work for single policies (average job completion time for AlloX, max-min fairness for Gandiva$_{\text{fair}}$). Moreover, Gandiva$_{\text{fair}}$ uses a second-price auction mechanism to improve the performance of a heterogeneity-agnostic max-min fairness scheme, but does not provide guarantees as to the optimality of the final allocation. On the other hand, Gavel formalizes each policy as an optimization problem, and can provide a guarantee that the returned solution is "optimal" according to the provided objective. Gavel is also able to support more sophisticated policies such as multi-level fairness.

Traditional Cluster Schedulers. Traditional schedulers such as Mesos [32], Borg [57], TetriSched [54], and YARN [56] support workloads with fixed heterogeneous resource requests, but do not reason about the diverse performance characteristics of jobs across accelerators. Mesos and YARN do not reason about interchangeable resource types that can run the same computation: for example, Mesos's DRF multi-resource sharing policy [26] decides how to give jobs allocations of distinct resource types, such as RAM and CPUs, but assumes that each job has declared which resources it needs to use and in what ratio (unlike our case, where we consider heterogeneity over accelerators themselves).

The multi-interchangeable resource allocation (MIRA) problem [53] also introduces the notion of effective throughput similar to Gavel, but does not demonstrate how this can be used to specify policies as optimization problems, does not consider performance optimizations like space sharing and placement sensitivity, and does not discuss how computed allocations can be realized on physical resources.

Omega [50], Apollo [16], and Hydra [20] are schedulers that take into account the fact that the target workload shows heterogeneity in the number and duration of constituent tasks. However, tasks largely take the same time on different CPUs, and heterogeneity in memory capacities only impacts the number and size of tasks that can be placed on a server. In our work, the compute devices themselves are interchangeable with sometimes large performance differences, and policies decide the time fractions of resources each job should receive while optimizing for various end objectives.

Dynamic Performance Estimation. As detailed in §6, Gavel uses the approach proposed by Quasar [21] to estimate co-located job performance online. In particular, Gavel uses a mix of profiling and matrix completion to compute a "fingerprint" against a set of reference models profiled offline. In this work, we show that the techniques used by Quasar can be successfully applied to this new setting.

Applicability to Other Settings. Even though we focused this paper on allocating heterogeneous resources for DNN training workloads, we believe that Gavel can be used for non-DNN workloads as well. Other workloads that are amenable to GPU execution, such as simulations, can be considered, even though performance estimates for these applications will be needed. We also believe the main technical insight presented in this paper – formulating diverse scheduling policies as optimization problems – is broadly applicable, and can be used to more easily deploy policies on homogeneous deep learning clusters, and on CPU clusters as well.

9 Conclusion

In this paper, we proposed Gavel, a heterogeneity-aware cluster scheduler that is able to optimize for many high-level metrics like fairness, makespan, and cost. Gavel demonstrates how existing policies can be expressed as optimization problems, and extends these policies to be heterogeneity-aware. Gavel then uses a decoupled round-based scheduling mechanism to ensure that the computed optimal allocation is realized. Gavel's heterogeneity-aware policies improve end objectives both on a physical and simulated cluster. It can support a higher average input job rate, while improving objectives such as average job completion time by 3.5×, makespan by 2.5×, and cost by 1.4×.

10 Acknowledgements

We thank our shepherd, Alexandra Fedorova, the anonymous OSDI reviewers, Firas Abuzaid, Trevor Gale, Shoumik Palkar, Deepti Raghavan, Daniel Kang, Pratiksha Thaker, and fellow Project Fiddle interns Jack Kosaian, Kshiteej Mahajan, and Jayashree Mohan for their invaluable feedback that made this work better. We thank MSR for their generous support of DN's and KS's internships, and for resources to develop and evaluate Gavel. This research was also supported in part by affiliate members and other supporters of the Stanford DAWN project— Ant Financial, Facebook, Google, Infosys, NEC, and VMware—as well as Toyota Research Institute, Northrop Grumman, Amazon Web Services, Cisco, NSF Graduate Research Fellowship grant DGE-1656518, and the NSF CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] AWS Accelerator Offerings. https://aws.amazon.com/ec2/instance-types/, 2020.

[2] Cloud GPUs on GCP. https://cloud.google.com/gpu, 2020.

[3] Cloud TPUs on GCP. https://cloud.google.com/tpu, 2020.

[4] gRPC. https://grpc.io, 2020.

[5] ImageNet Training in PyTorch. https://github.com/pytorch/examples/tree/master/imagenet, 2020.

[6] Implementing Core Scheduler Functionality in Resource Manager (V1) for Hadoop. https://issues.apache.org/jira/browse/HADOOP-3445, 2020.

[7] Job Scheduling in Spark. https://spark.apache.org/docs/latest/job-scheduling.html#scheduling-within-an-application, 2020.

[8] Linear-fractional Optimization. http://www.seas.ucla.edu/~vandenbe/ee236a/lectures/lfp.pdf, 2020.

[9] Microsoft Philly Trace. https://github.com/msr-fiddle/philly-traces, 2020.

[10] NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020.

[11] Word-level Language Modeling RNN. https://github.com/pytorch/examples/tree/master/word_language_model, 2020.

[12] YARN – The Capacity Scheduler. https://blog.cloudera.com/yarn-capacity-scheduler/, 2020.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[14] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[15] D. P. Bertsekas and R. G. Gallager. Data Networks. 1987.

[16] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.

[17] E. J. Candes and Y. Plan. Matrix Completion with Noise. *Proceedings of the IEEE*, 98(6):925–936, 2010.

[18] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[19] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of DAWNBench, A Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Operating Systems Review*, 53(1):14–25, 2019.

[20] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, et al. Hydra: A Federated Resource Manager for Data-Center Scale Analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, 2019.

[21] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 127–144, 2014.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[23] S. Diamond and S. Boyd. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[24] D. Elliott, S. Frank, K. Sima'an, and L. Specia. Multi30K: Multilingual English-German Image Descriptions. In *Proceedings of the 5th Workshop on Vision and Language*, pages 70–74. Association for Computational Linguistics, 2016.

[25] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[26] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, pages 24–24, 2011.

[27] D. Griffis. RL A3C PyTorch. https://github.com/dgriff777/rl_a3c_pytorch, 2020.

[28] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[29] F. M. Harper and J. A. Konstan. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TIIS)*, 5(4):19, 2016.

[30] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2961–2969, 2017.

[31] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[32] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, pages 22–22, 2011.

[33] Y.-H. Huang. Attention is All You Need: A PyTorch Implementation. https://github.com/jadore801120/attention-is-all-you-need-pytorch, 2018.

[34] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX Annual Technical Conference, USENIX ATC 2019*, pages 947–960, 2019.

[35] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[36] A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-10 Dataset. http://www.cs.toronto.edu/kriz/cifar.html, 2014.

[37] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu. AlloX: Compute Allocation in Hybrid Clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[38] E. Linder-Norén. PyTorch-GAN. https://github.com/eriklindernoren/PyTorch-GAN#cyclegan, 2020.

[39] K. Liu. Train CIFAR-10 with PyTorch. https://github.com/kuangliu/pytorch-cifar, 2020.

[40] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[41] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[42] S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer Sentinel Mixture Models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[43] A. Mnih and R. R. Salakhutdinov. Probabilistic Matrix Factorization. In *Advances in Neural Information Processing Systems*, pages 1257–1264, 2008.

[44] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[45] A. Moussawi. Towards Large Scale Training of Autoencoders for Collaborative Filtering. In *Proceedings of Late-Breaking Results Track Part of the Twelfth ACM Conference on Recommender Systems*, RecSys'18, Vancouver, BC, Canada, 2018.

[46] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[47] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning (December 2018)*, 2018.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[49] B. Radunovic and J.-Y. Le Boudec. A Unified Framework for Max-Min and Min-Max Fairness with Applications. *IEEE/ACM Transactions on Networking*, 15(5):1073–1083, 2007.

[50] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[51] M. J. Shafiee, B. Chywl, F. Li, and A. Wong. Fast YOLO: A Fast You Only Look Once System for Real-Time Embedded Object Detection in Video. *arXiv preprint arXiv:1709.05943*, 2017.

[52] P. Sinha and A. A. Zoltners. The Multiple-Choice Knapsack Problem. *Operations Research*, 27(3):503–515, 1979.

[53] X. Sun, T. N. Le, M. Chowdhury, and Z. Liu. Fair Allocation of Heterogeneous and Interchangeable Resources. *ACM SIGMETRICS Performance Evaluation Review*, 46(2):21–23, 2019.

[54] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 35. ACM, 2016.

[55] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[56] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[57] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18, 2015.

[58] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[59] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, pages 265–278. ACM, 2010.

[60] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2223–2232, 2017.

A Artifact Appendix

A.1 Abstract

Gavel is open sourced at https://github.com/stanford-futuredata/gavel. We provide implementations for Gavel's heterogeneity-aware policies, its round-based scheduling mechanism, and the `GavelIterator` interface, as well as implementations of relevant baselines such as AlloX [37], a simulator, and code to reproduce the graphs and other quantitative results shown in this paper.

A.2 Artifact check-list

- **Algorithm:** Heterogeneity-aware policies are expressed as optimization problems over allocations. Scheduling is performed using a greedy round-based scheduling mechanism.
- **Hardware:** Experiments in simulation can run on a multi-core server with Ubuntu 16.04. Experiments on a physical cluster need Nvidia GPUs.
- **Setup instructions:** Setup instructions are available in the `README.md` and `EXPERIMENTS.md` files provided in the artifact.
- **Experiments:** All results presented in this paper can be reproduced using the provided artifact.
- **Required disk space:** About 100 GB for logfiles when running simulated cluster experiments, about 10 GB for intermediate model checkpoints for physical cluster experiments, about 150 GB for datasets.
- **Expected experiment run time:** Days to a week for full simulated experiments, shorter durations (hours to a day) for scaled-down experiments (smaller cluster and trace).
- **Public link:** https://github.com/stanford-futuredata/gavel.
- **Code licenses:** MIT License.

A.3 Description

A.3.1 How to access

The artifact is publicly available at https://github.com/stanford-futuredata/gavel.

A.3.2 Hardware dependencies

Simulated experiments can be run on any multicore server. We ran experiments on a 56-core server with Ubuntu 16.04. Physical clusters need to have Nvidia GPU accelerators; other accelerators supported by Deep Learning frameworks such as PyTorch are supported as well by the scheduler.

A.3.3 Software dependencies

Software dependencies are specified at https://github.com/stanford-futuredata/gavel/blob/master/README.md.

A.3.4 Datasets

Running the simulator does not require any external datasets. When running physical cluster experiments, training data for training jobs is needed. These are task-specific (for example, image classification training jobs might use the ImageNet dataset).

A.4 Installation

Installation instructions are specified at https://github.com/stanford-futuredata/gavel/blob/master/README.md.

A.5 Experiment workflow

Experiments in simulation are triggered by a driver script that instantiates the scheduler, and then adds jobs to the simulated cluster either according to a pre-defined trace, or on-the-fly using distributions with input parameters specified by the user. The scheduler computes the optimal allocation for each active job based on the desired policy and target objective, and then assigns resources to jobs according to this computed allocation using its round-based scheduling mechanism. Oracle throughputs are used to estimate the progress of jobs given a specified amount of time on the given resources. At the end of a run, completion times of all jobs of interest are recorded. Jobs of interest are usually a subset of all jobs submitted to the cluster, since we want to study steady state behavior. An exception is made for makespan policies, which try to minimize the total time taken by a collection of jobs; for this policy, jobs are added once at the start of the trace, and then jobs are allowed to drain from the cluster.

Experiments on physical clusters are also triggered by a driver script run on the scheduler, but are different in one key aspect: jobs are run on real accelerators for the specified number of steps. Every round, the scheduler makes a scheduling decision to decide what resources should be given to the different jobs. As before, job completion times are recorded when a job finishes executing.

A.6 Evaluation and expected result

Each experiment run results in an output logfile that records the microtasks run every scheduling round, as well as the completion times for each job. These logfiles can then be parsed to produce the graphs and other quantitative results presented in the evaluation section of this paper. Code to parse and produce plots are available at https://github.com/stanford-futuredata/gavel/tree/master/scheduler/notebooks/figures.

A.7 Experiment customization

Experiments can be run with different seeds using the main sweep scripts. Experiments can also be scaled down in different ways to obtain results faster: a) smaller cluster, b) fewer traces, c) smaller traces, and d) smaller set of jobs of interest over which objectives (such as average JCT) are measured.

A.8 AE Methodology

Submission, reviewing and badging methodology is specified at https://www.usenix.org/conference/osdi20/call-for-artifacts.

PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications

Zhihao Bai* Zhen Zhang* Yibo Zhu† Xin Jin*

*Johns Hopkins University †ByteDance Inc.

Abstract

Deep learning (DL) workloads include throughput-intensive training tasks and latency-sensitive inference tasks. The dominant practice today is to provision dedicated GPU clusters for training and inference separately. Due to the need to meet strict Service-Level Objectives (SLOs), GPU clusters are often over-provisioned based on the peak load with limited sharing between applications and task types.

We present PipeSwitch, a system that enables unused cycles of an inference application to be filled by training or other inference applications. It allows multiple DL applications to *time-share* the same GPU with the entire GPU memory and *millisecond-scale* switching overhead. With PipeSwitch, GPU utilization can be significantly improved without sacrificing SLOs. We achieve so by introducing *pipelined context switching*. The key idea is to leverage the *layered* structure of neural network models and their *layer-by-layer* computation pattern to *pipeline* model transmission over the PCIe and task execution in the GPU with model-aware grouping. We also design unified memory management and active-standby worker switching mechanisms to accompany the pipelining and ensure process-level isolation. We have built a PipeSwitch prototype and integrated it with PyTorch. Experiments on a variety of DL models and GPU cards show that PipeSwitch only incurs a task startup overhead of 3.6–6.6 ms and a total overhead of 5.4–34.6 ms (10–50× better than NVIDIA MPS), and achieves near 100% GPU utilization.

1 Introduction

Deep learning (DL) powers an emerging family of intelligent applications in many domains, from retail and transportation, to finance and healthcare. GPUs are one of the most widely-used classes of accelerators for DL. They provide better trade-off between performance, cost and energy consumption than CPUs for deep neural network (DNN) models.

DL workloads include throughput-intensive training tasks and latency-sensitive inference tasks. The dominant practice today is to provision dedicated GPU clusters for training and inference separately. Inference tasks cannot be served with training clusters under flash crowds, and training tasks cannot utilize inference clusters when the inference load is low. Consequently, inference clusters are often over-provisioned for the peak load, in order to meet strict Service Level Objectives (SLOs). Even for inference itself, production systems are typically provisioned to each application on per-GPU granularity to limit the interference between applications.

Ideally, multiple DL applications should be able to be packed to the same GPU server to maximize GPU utilization via time-sharing. This is exactly how operating systems achieve high CPU utilization via task scheduling and context switching. The idea of fine-grained CPU time-sharing has been further extended to cluster scheduling. For example, Google Borg [1] packs online services and batch jobs, and saves 20%-30% machines (compared with not packing them). Why can't we use GPUs in the same way?

The gap is that GPU has high overhead when switching between tasks. Consequently, naively using GPUs in the same way as CPUs will not satisfy the requirements of DL inference that have strict SLOs in the range of tens to hundreds of milliseconds [2, 3]. If a GPU switches to a DNN model (*e.g.,* ResNet) that has not been preloaded onto the GPU, it can take multiple seconds before serving the first inference request, even with state-of-the-art tricks like CUDA unified memory [4] (§6). In contrast, CPU applications can be switched in milliseconds or even microseconds [5].

To avoid such switching overhead, the existing solution is to spatially share the GPU memory. For example, although NVIDIA Multiple Process Sharing (MPS) [6] and Salus [7] allow multiple processes to use the same GPU, they require all processes' data (e.g., DNN models) to be preloaded into the GPU memory. Unfortunately, the GPU memory is much more limited than host memory and cannot preload many applications. Sometimes, just one single memory-intensive training task may consume all the GPU memory. Moreover, the memory footprints of inference tasks are also increasing—the models are getting larger, and request batching is prevalently used to increase throughput [3]. In addition, this approach does not provide strong GPU memory isolation between applications.

As such, we argue that a context switching design that minimizes the switching overhead, especially quickly switching the contents on GPU memory, is a better approach for efficiently time-sharing GPUs. The DNN models can be held in host memory, which is much larger and cheaper than GPU memory, and the GPU can quickly context-switch between

the models either for training or inference. This way, *the number of* applications that can be multiplexed is not limited by the GPU memory size, and each application is able to use the entire GPU compute and memory resources during its time slice. To our best knowledge, no existing solution offers such context switching abstraction for GPU.

To this end, we propose PipeSwitch, a system that (i) enables GPU-efficient multiplexing of many DL applications on GPU servers via fine-grained time-sharing, and (ii) achieves millisecond-scale latencies and high throughput as dedicated servers. PipeSwitch enables unused cycles of an inference application to be filled by training or other inference applications. We achieve so by introducing a new technology called *pipelined context switching* that exploits the characteristics of DL applications to achieve millisecond-scale overhead for switching tasks on GPUs. Such small switching overhead is critical for DL applications to satisfy strict SLO requirements.

To understand the problem, we first perform a measurement study to profile the task switching overhead and analyze the overhead of each component. We divide the switching overhead into four components, which are old task cleaning, new task initialization, GPU memory allocation, and model transmission via PCIe from CPU to GPU. Every component takes a considerable amount of time, varying from tens of milliseconds to seconds. Such overhead is significant, because an inference task itself only takes tens of milliseconds on a GPU and the latency SLOs are typically a small multiple of the inference time [3].

We take a holistic approach, and exploit the characteristics of DL applications to minimize the overhead of all the components. Our design is based on a key observation that DNN models have a *layered* structure and a *layer-by-layer* computation pattern. As such, there is no need to wait for the entire model to be transmitted to the GPU before starting computation. Based on this observation, we design a pipelined model transmission mechanism, which *pipelines* model transmission over the PCIe and model computation in the GPU. Naive pipelining on per-layer granularity introduces high overhead on tensor transmission and synchronization. We divide layers into groups, and design an optimal model-aware grouping algorithm to find the best grouping strategy for a given model.

The computation of a DL task is layer by layer, which has a simple, regular pattern for memory allocation. The default general-purpose GPU memory management (e.g., CUDA unified memory [4]) is an overkill and incurs unnecessary overhead. We design unified memory management with a dedicated memory daemon to minimize the overhead. The daemon pre-allocates the GPU memory, and re-allocates it to each task, without involving the expensive GPU memory manager. The DNN models are stored only once in the memory daemon, instead of in every worker, to minimize memory footprint. We exploit that the memory allocation for a DNN model is deterministic to eliminate extra memory copies between the daemon and the workers and reduce the IPC overhead .

We use an active-standby mechanism for fast worker switching and process-level isolation. Each server contains an active worker and multiple standby workers. The active worker executes the current task on the GPU; the standby workers stay on the CPU and wait for the next task. Our mechanism parallelizes old task cleaning in the active worker and new task initialization in the standby worker to minimize worker switching overhead. With separate worker processes, PipeSwitch enforces process-level isolation.

Pipelining is a canonical technique widely used in computer systems to improve system performance and maximize resource utilization. Prior work in DL systems such as PipeDream [8] and ByteScheduler [9] has applied pipelining to distributed training. These solutions focus on *inter-batch* pipelining to overlap computation and *gradient* transmission of different batches for *training* workloads of the *same* DNN model. The key novelty of PipeSwitch is that it introduces *intra-batch* pipelining to overlap *model* transmission and computation to reduce the overhead of switching between *different* DNN models, which can be *either inference or training*. Unlike pipelining for the same task, PipeSwitch requires us to address new technical challenges on memory management and worker switching across *different* processes. We design new techniques to not only support training, but also inference that has strict SLOs.

In summary, we make the following contributions.

- We propose PipeSwitch, a system that enables GPU-efficient fine-grained time-sharing for multiple DL applications, and achieves millisecond-scale context switching latencies and high throughput.
- We introduce *pipelined context switching*, which exploits the characteristics of DL applications, and leverages pipelined model transmission, unified memory management, and active-standby worker switching to minimize switching overhead and enforce process-level isolation.
- We implement a system prototype and integrate it with PyTorch. Experiments on a variety of DL models and GPU cards show that PipeSwitch only incurs a task startup overhead of 3.6–6.6 ms and a total overhead of 5.4–34.6 ms (10–50× better than NVIDIA MPS), and achieves near 100% GPU utilization.

2 Motivation

In this section, we identify the inefficiencies in today's shared GPU clusters, and motivate running DL workloads on GPUs in the fine-grained time-sharing model.

2.1 GPU Clusters

Shared GPU clusters. To run DNN workloads in a large scale, enterprises build GPU clusters that are either privately [10, 11] or publicly [12–14] shared by multiple users. Such GPU clusters are usually specifically designed with dedicated physical forms and power supplies, along with high speed networks and specialized task schedulers.

Why build a shared cluster instead of a dedicated one for each user? The main reason is to bring down the cost. The demand of training is not well predictable—it would depend on the progress of different developers. The demand of inference is more predictable, e.g., an inference task for a particular application usually has a daily periodical pattern based on the application usage. Nevertheless, the patterns can still vary across different tasks. Like traditional CPU workloads, a shared cluster by different tasks would increase the resource utilization via time-sharing.

No sharing between training and inference. However, such "shared" clusters are not shared between training and inference. Even though training and inference both use GPUs, the current practice is to build dedicated clusters for training and inference separately. This brings several inefficiencies.

- Inference clusters are over-provisioned for the peak load, because they directly serve user requests and need to meet strict SLOs. Although inference clusters are not always running at high utilization, they cannot be utilized by training.
- Training clusters are equipped with powerful GPUs to run training tasks, which are often elastic and do not have strict deadlines. However, when there is a flash crowd (e.g., an application suddenly becomes popular and the demand grows beyond the operator's expectation), the training cluster cannot preempt the training tasks for inference tasks.
- Even for inference tasks, production systems often allocate GPUs to applications on per-GPU granularity (e.g., binding GPUs to the VMs, containers or processes of an application), in order to limit the interference between different applications and satisfy the SLO requirements.

One of the reasons for separately provisioning is that GPUs designed for inference tasks might be too wimpy for training tasks. This, however, has started to change with the arrival of new GPU hardware, most notably NVIDIA T4. Compared with NVIDIA V100 which has up to 32GB GPU memory and 15.7 TFLOPS (single-precision), NVIDIA T4 has comparable performance with 16GB GPU memory and 8.1 TFLOPS (single-precision). Also, new algorithms and systems for distributed training [8, 9, 15, 16] enable multiple GPUs to accelerate training, if one GPU is not fast enough.

Our industry collaborator, a leading online service provider, confirms this observation. This service provider currently runs more than 10K V100 GPUs for training, and at least 5× as many T4 GPUs for inference. The computation power on both sides is within the same order of magnitude. The inference workload fluctuates in correlation with the number of active users, and shows clear peaks and valleys within each day—the peak demand during daytime is $> 2\times$ of the valley at midnight. It would be a great match to utilize inference GPUs during less busy times for training models that require daily updates with latest data. A good example is to fine-tune BERT using daily news. This means great opportunity in improving GPU utilization by Borg-like [1] systems for GPUs.

2.2 Fine-Grained Time-Sharing GPU

We envision to build GPU clusters that can be shared across different applications including training and inference. We propose to pack multiple DL applications onto the same GPU via fine-grained time-sharing abstraction to maximize GPU utilization. This is inspired by the OS scheduler and context switching in the CPU world. It has the following advantages.

- It would dramatically improve the resource utilization, especially because inference and training workloads have complementary usage patterns. Online inference services are often more idle during midnight, while many training developers would start a time-consuming job at night. Besides, inference loads on different models have different patterns, which also benefits from the time sharing.
- Similar to CPU workloads, fine-grained time-sharing can provide better utilization than provisioning dedicated resources, while providing necessary process-level isolation.
- It would greatly simplify the design of load balancers and schedulers as any server would be able to run any task with low overhead to switch between different applications.

The gap: the precious GPU memory and slow switching. To achieve this goal, however, we face a major challenge—fast GPU context switching between different processes. A modern server can be equipped with several TB of host memory, enabling it to load many applications. However, task execution on GPUs require GPU memory, which is very limited even on high-end GPUs, e.g., 16 GB for T4 and 32 GB for V100. More importantly, GPU memory is purposed for task execution, not for storing the state of idle applications. DL tasks, especially training, require a large amount, or even all of the memory on a GPU.

Storing the models in the GPU like Salus [7] cannot support training tasks which are memory-intensive or even multiple inference tasks which have large models. This is particularly important as state-of-the-art models are getting deeper and larger, and thus even idle applications can occupy large memory space. In addition, request batching is prevalently used to increase throughput [3], which further increases the GPU memory requirement of inference applications. Ideally, the active application should be able to utilize the entire GPU memory for its purpose, and the number of applications that can be served by a GPU server should only be limited by its host memory size. Consequently, switching a task would require heavy memory swapping.

Unfortunately, many online inference workloads require strict SLOs that naive memory swapping between the host memory and the GPU memory cannot meet. For example, we test the strawman scenario where we stop a training task and then start an inference task. The first inference batch would require several seconds to finish (§4.1). Existing support such as NVIDIA MPS is not optimized for DL workloads, and incurs hundreds of milliseconds overhead (§6).

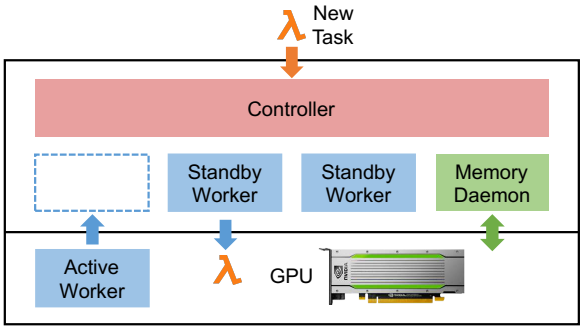


Figure 1: PipeSwitch architecture.

The opportunity: DL workloads have well-defined structures. Fortunately, the structure and computation pattern of DNN models allow us to highly optimize task switching and achieve millisecond-scale overhead. DNN models are usually *deep*, consisting of multiple layers stacking one on another. Furthermore, the computation of DNN models takes place layer by layer as well. Thus, it is possible to build a pipeline that overlaps the computation and GPU memory swapping for fast context switching.

In the following sections, we will show that such pipeline is indeed feasible and effective. In addition, we will also need to resolve other challenges like memory management and worker switching. Combining all the ideas into our system, PipeSwitch, we close the gap of GPU memory sharing and switching, and enable the design of an efficient time-sharing GPU cluster for DL workloads.

3 PipeSwitch Overview

PipeSwitch enables GPU-efficient multiplexing of multiple DL applications on GPU servers. It exploits the characteristics of DL applications to achieve *millisecond-scale* task switching overhead in order to satisfy SLO requirements. Such fast task switching enables more flexible fine-grained scheduling to improve GPU utilization for dynamic workloads. It benefits switching not only between inference and training, but also between inference on different models. Here we provide an overview of the architecture and task execution.

System architecture. Figure 1 shows the architecture of a PipeSwitch server. This server contains four types of components: a controller, a memory daemon, an active worker, and multiple standby workers.

- **Controller.** The controller is the central component. It receives tasks from clients, and controls the memory daemon and the workers to execute the tasks.
- **Memory daemon.** The memory daemon manages the GPU memory and the DNN models. It allocates the GPU memory to the active worker, and transfers the model from the host memory to the GPU memory.
- **Active worker.** The active worker is the worker that currently executes a task in the GPU. Here a worker is a process that executes tasks on one GPU.

Instance Type	g4dn.2xlarge	p3.2xlarge
GPU Type	NVIDIA T4	NVIDIA V100
Task Cleaning	155 ms	165 ms
Task Initialization	5530 ms	7290 ms
Memory Allocation	10 ms	13 ms
Model Transmission	91 ms	81 ms
Total Overhead	5787 ms	7551 ms
Inference Time	105 ms	32 ms

Table 1: Measurement results of task switching overhead and the breakdown of individual components. *All* components should be optimized to meet the SLOs.

- **Standby worker.** A server has one or more standby workers. A standby worker is idle, is initializing a new task, or is cleaning its environment for the previous task.

Task execution. The controller queues a set of tasks received from the clients. It uses a scheduling policy to decide which task to execute next. It supports canonical scheduling policies such as first come first serve (FCFS) and earliest deadline first (EDF), and can be easily extended to support new policies. We focus on fast context switching, and the specific scheduling algorithm is orthogonal to this paper. The scheduling is *preemptive*, i.e., the controller can preempt the current task for the next one based on the scheduling policy. For example, if the current task is a training task, the controller can preempt it for an inference task that has a strict latency SLO.

To start a new task, the controller either waits for the current task to finish (e.g., if it is inference) or preempts it by notifying the active worker to stop (e.g., if it is training). At the same time, the controller notifies an idle standby worker to initialize its environment for the new task. After the active worker completes or stops the current task, the controller notifies the memory daemon and the standby worker to load the task to GPU to execute with pipelined model transmission (§4.2). The memory daemon allocates the memory to the standby worker (§4.3), and transmits the model used by the new task from the host memory to the GPU memory. The standby worker becomes the new active worker to execute the new task, and the active worker becomes a standby worker and cleans the environment for the previous task (§4.4). The primary goal of this paper is to design a set of techniques based on the characteristics of DL applications to minimize the task switching overhead in this process.

4 PipeSwitch Design

We first perform a measurement study to profile the task switching overhead and break it down to individual components. Then we describe our design to systematically minimize the overhead of each component.

4.1 Profiling Task Switching Overhead

In order to understand the problem, we perform a measurement study to profile the task switching overhead. The mea-

surement considers a typical scenario that a server stops a training task running on the GPU, and then starts an inference task. The DNN model used in the measurement is ResNet152 [17]. The measurement covers two types of instances on Amazon AWS, which are g4dn.2xlarge with NVIDA T4 and p3.2xlarge with NVIDIA V100. We assume the inference task has arrived at the server, and focus on measuring the time to start and execute it on the GPU. We exclude the network time and the task queueing time.

Table 1 shows the results. The total times to start the inference task on the GPUs are 5787 ms and 7551 ms, respectively. We break the overhead down into the four components.

- **Task cleaning.** The training task stops and cleans its GPU environment, such as freeing the GPU memory.
- **Task initialization.** The inference task creates and initializes its environment (i.e., process launching, PyTorch CUDA runtime loading, and CUDA context initialization).
- **Memory allocation.** The inference task allocates GPU memory for its neural network model.
- **Model transmission.** The inference task transmits the model from the host memory to the GPU memory.

The inference time on V100 is lower than that on T4, and both of them are significantly lower than the total overheads. The reason for lower overhead on T4 is that task switching largely depends on CPU, and g4dn.2xlarge is equipped with better CPU than p3.2xlarge (Intel Platinum 8259CL vs. Intel Xeon E5-2686 v4). A strawman solution that simply stops the old task and starts the new task would easily violate SLOs.

Because all the components take considerable time compared to the inference time, we emphasize that *all* the components should be optimized to achieve minimal switching overhead and meet the SLOs.

4.2 Pipelined Model Transmission

Transmitting a task from CPU to GPU is bounded by the PCIe bandwidth. The PCIe bandwidth is the physical limit on how fast an arbitrary task can be loaded to the GPU. We exploit the characteristics of DL applications to *circumvent* this physical limit. Our key observation is that DNN models have a *layered* structure. The computation is performed layer by layer. An inference task only performs a forward pass from the first layer to the final layer to make a prediction; each iteration in a training task performs a forward pass and then a backward pass. In both cases, a task does not need to wait for the entire model to be transmitted to the GPU before beginning the computation. Instead, the task can start the computation of a layer as soon as the layer is loaded in the GPU and the input of the layer is ready (i.e., the previous layers have finished their computation), regardless of its following layers. Figure 2 illustrates the advantage of pipelining over the strawman solution.

PipeSwitch requires the knowledge of models. PipeSwitch does not modify the model structure, and only adds hooks for PyTorch to wait for transmission or synchronize the execution.

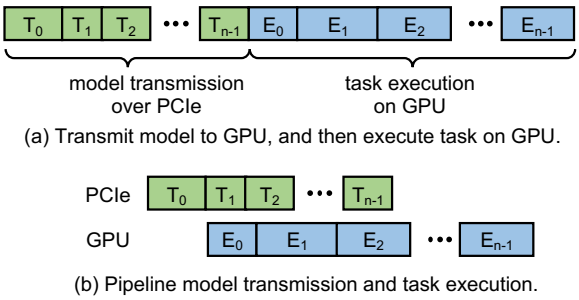


Figure 2: PipeSwitch pipelines model transmission and task execution. The example shows an inference task that only has a forward pass in task execution.

Adding hooks can be automated, and PipeSwitch can be implemented as a part of the DNN framework, e.g., PyTorch, so it can gather the model structure information while remaining transparent to users and cluster managers.

Optimal model-aware grouping. The basic way for pipelining is to pipeline on *per-layer* granularity, i.e., the system transmits the layers to the GPU memory one by one, and the computation for a layer is blocked before the layer is transmitted. Pipelining brings two sources of system overheads. One is the overhead to invoke multiple calls to PCIe to transmit the data. For a large amount of data (e.g., combining the entire model to a large tensor to transmit together), the transmission overhead is dominated by the data size. But when we divide the model into many layers, invoking a PCIe call for each layer, especially given that some layers can be very small, would cause significant extra overhead. The other is the synchronization overhead between transmission and computation, which is necessary for the computation to know when a layer is ready to compute. Pipelining on per-layer granularity requires synchronization for every layer.

We use grouping to minimize these two sources of overhead. We combine multiple layers into a group, and the pipelining is performed on *per-group* granularity. In this way, the pipelining overhead is paid once for each group, instead of each layer. Grouping introduces a trade-off between pipelining efficiency and pipelining overhead. On one hand, using small groups (e.g., per-layer in the extreme case) enables more overlap between transmission and computation, which improves pipelining efficiency, but it also pays more pipelining overhead. On the other hand, using big groups (e.g., the entire model in one group in the extreme case) has minimal pipelining overhead, but reduces the chance for overlapping.

Grouping must be model-aware, because models have different structures in terms of the number of layers and the size of each layer. Naively, we can enumerate all possible combinations to find the optimal grouping strategy. This is not amenable because large models can have hundreds of layers and the time complexity for enumeration is exponential.

In order to find the optimal grouping strategy efficiently, we introduce two pruning techniques based on two insights.

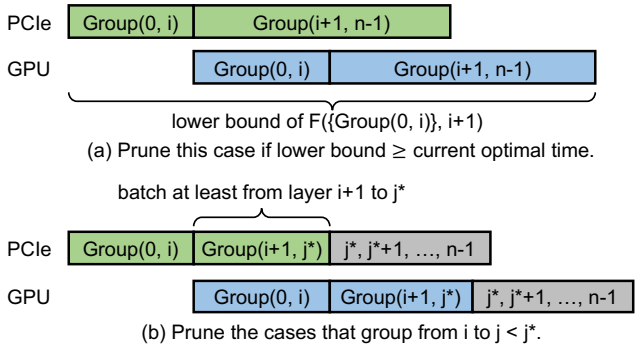


Figure 3: Examples for two pruning techniques.

Before we dive into the details, we first formulate the problem. Let the number of layers be n. Let $F(B,i)$ be a function that returns the total time of the optimal grouping strategy from layer i to n-1 given that layer 0 to i-1 have formed groups represented by B. Then we have the following recursive formula.

$$F(\{\},0) = \min_i F(\{group(0,i)\},i+1) \quad (1)$$

Specifically, to find the optimal grouping strategy for the entire model (i.e., F({}, 0)), we divide all possible combinations into n cases based on how the first group is formed, i.e., case i means the first group contains layer 0 to i. This formula can be applied recursively to compute $F(\{group(0,i)\},i+1)$.

Our first insight is that it is not necessary to examine all the n cases, because if the first group contains too many layers, the computation of the first group would be delayed too much to compensate the pipeline efficiency. Let $T(i,j)$ and $E(i,j)$ be the transmission and execution times for a group from layer i to j respectively, where $T(i,j)$ is calculated based on the size of layer i to j and PCIe bandwidth, and $E(i,j)$ is profiled on the GPU. Note that the overhead of invoking multiple calls is included in $T(i,j)$. As illustrated by Figure 3(a), we compute a lower bound for the total time for each case in Equation 1.

$$\begin{aligned} F(\{group(0,i)\},i+1) \geq \\ \min(T(0,i)+T(i+1,n-1), \\ T(0,i)+E(0,i)+E(i+1,n-1)) \end{aligned} \quad (2)$$

The lower bound considers the best case that all the remaining layers are combined in one group for transmission and computation, and that the computation and communication can be perfectly overlapped, i.e., its computation can happen right after the computation of the first group finishes. If the lower bound of case i is already larger than the total time of the best grouping strategy found so far, then case i (i.e., the recursive computation for $F(\{group(0,i)\},i+1)$) can be pruned.

Our second insight is that other than the first group, we can safely pack multiple layers in a group based on the progress of computation without affecting pipeline efficiency. Figure 3(b) shows an example for this insight. Suppose that we have already fixed the first group to be from layer 0 to i, and we apply Equation 1 recursively to enumerate the cases for the second group. We can hide the transmission of the second group into the computation of the first group, as long as the transmission finishes no later than the computation of the first group. The least number of layers to group can be computed using the following equation.

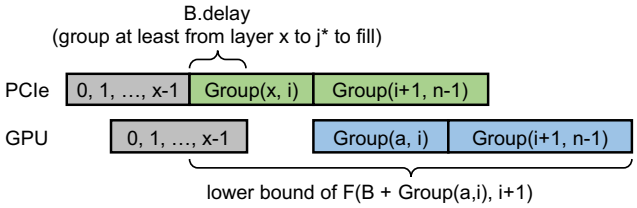


Figure 4: General case for the two pruning techniques.

$$j^* = \arg\max_j T(i+1,j) \leq E(0,i) \quad (3)$$

Group from layer $(i+1)$ to $j < j^*$ is no better than grouping from $(i+1)$ to j^* because it does not increase the pipeline efficiency and has higher pipeline overhead. Therefore, we can prune the cases that group from layer $(i+1)$ to $j < j^*$ and only search for $j \geq j^*$.

Algorithm. Based on these two insights, we design an algorithm to find the optimal grouping strategy for a given model. We emphasize that this algorithm runs *offline* to find the strategy, and the resulting strategy is used online by PipeSwitch for context switching. Algorithm 1 shows the pseudo code. The function *FindOptGrouping* recursively finds the optimal grouping strategy based on Equation 1 (line 1-27). It takes two inputs: B represents the groups that have already formed, x is the first layer that have not formed a group. It uses *opt_groups* to store the best grouping strategy from layer x given B, which is initialized to none (line 2). The algorithm applies the second pruning insight to form the first group from layer x (line 3-9). Equation 3 and Figure 3(b) illustrate this insight with a special example that B only contains one group from layer 0 to i. In general, B can contain multiple groups formed by previous layers, and we use $B.delay$ to denote the time to which the group can be formed, as shown in Figure 4. The algorithm finds j^* based on $B.delay$ (line 4-9), and the enumeration for i can skip the layers from x to j^*-1 (line 11). For case i, the algorithm applies the first insight to compute the lower bound (line 12-17). Again, the example in Equation 2 and Figure 3(a) is a special case when x is 0. For the general case, the computation from x has to wait for both its transmission (i.e., $T(x,i)$) and the computation of the previous groups (i.e., $B.delay$), as shown in Figure 4. If the lower bound is already bigger than the current optimal time, then case i is pruned (line 18-19). Given the group from layer x to i is formed, the function recursively applies itself to find the optimal groups from layer i+1 to n-1 (line 21-23), and updates *opt_groups* if the current strategy is better (line 24-26). Finally, it returns *opt_groups* (line 27). In practice, we use a heuristic that bootstraps *opt_groups* with a relative

Algorithm 1 Optimal Model-Aware Grouping

```
1:  function FindOptGrouping(B, x)
2:      opt_groups ← ∅, opt_groups.time ← ∞
3:      // find first group from layer i to j*
4:      j* ← x
5:      for layer i from x to n − 1 do
6:          if T(x,i) ≤ B.delay then
7:              j* ← i
8:          else
9:              break
10:     // recursively find the optimal grouping
11:     for layer i from j* to n − 1 do
12:         if opt_groups ≠ ∅ then
13:             // compute lower bound
14:             trans_time ← T(x,i) + T(i+1,n−1)
15:             exec_time ← max(T(x,i), B.delay)
16:                         + E(x,i) + E(i+1,n−1)
17:             lower_bound ← min(trans_time, exec_time)
18:             if lower_bound > opt_groups.time then
19:                 continue
20:         // recursively find rest groups
21:         first_group ← Group(x,i)
22:         rest_groups ← FindOptGrouping(
23:                         B + first_group, i+1)
24:         cur_groups ← first_group + rest_groups
25:         if cur_groups.time < opt_groups.time then
26:             opt_groups ← cur_groups
27:     return opt_groups
```

good strategy (e.g., group every ten layers). Given n layers, there are 2^{n-1} different grouping strategies, so the time complexity of Algorithm 1 is $O(2^n)$, as in the worst case it needs to enumerate all strategies. The two pruning techniques are able to prune most of the strategies, and can quickly find the optimal one as we will show in §6. We have the following theorem for the algorithm.

Theorem 1. *Algorithm 1 finds the optimal grouping strategy that minimizes the total time for the pipeline.*

Proof. Algorithm 1 computes the recursive function $FindOptGrouping(B,x)$. Let $m = n - x$, which is the number of layers the function considers. We use induction on m to show that $FindOptGrouping(B,x)$ outputs the optimal grouping strategy from layer x to $n-1$ given that previous layers have formed groups represented by B.

Base case. When $m = 1$, the function only examines one layer. Because there is only one strategy which is layer x itself is one group, this strategy is the optimal strategy.

Inductive step. Assume that for some $k \geq 1$ and any $m \leq k$, $FindOptGrouping(B,x)$ outputs the optimal strategy. Consider $m = k+1$, i.e., the algorithm now considers $k+1$ layers. The algorithm divides the problem into $k+1$ cases, where case i ($0 \leq i \leq k$) forms the first group from layer x to $x+i$.

For case i where $0 \leq i \leq k-1$, because $FindOptGrouping(B + Group(x,x+i), x+i+1)$ only considers $k - i \leq k$ layers, it outputs the optimal grouping strategy for case i based on the assumption.

For case $i = k$, the first group contains all layers from x to $n-1$. The optimal strategy for this case is one group.

Because these cases are exclusive and cover the entire search space, by choosing the optimal grouping strategy from these cases, the algorithm outputs the optimal grouping strategy for $m = k+1$.

The algorithm uses two pruning techniques. The first technique prunes the cases if their lower bounds are no better than the current found optimal. It is obvious that this technique does not affect the optimality. The second technique prunes the case if their first groups are from layer x to $j < j^*$. Because these cases cannot advance the computation to an earlier point than grouping from x to at least j^*, pruning these cases also do not affect the optimality. □

Generality. Algorithm 1 achieves optimality for a given list of layers. This, however, does not require the models to be linear. In general, the layers or operators in a DNN model can be connected as an arbitrary computation graph, instead of a simple chain. Models like ResNet and Inception are technically non-linear directed acyclic graph (DAGs). Yet, there is an execution order that the layers/operators in the DAG are issued to the GPU one by one. Algorithm 1 does not have any special assumptions on the execution order. It is only interested in finding out how to group the layers given the execution order (and corresponding data dependencies) to achieve high pipelining efficiency and low pipelining overhead. It even applies for graphs with loops, in which the order is based on the first time an operator is executed. The order does not affect correctness, because an operator can be executed only when it is transmitted to the GPU and the input is ready. Thus, our pipelined model transmission is applicable to the general case.

4.3 Unified Memory Management

Task execution in a GPU requires GPU memory. A GPU has its own memory management system, and provides a `malloc` function (e.g., `cudaMalloc` for NVIDIA GPUs) similar to CPUs for memory allocation. NVIDIA also provides CUDA unified memory [4] to automatically handle memory movement between the host memory and the GPU memory for applications. A naive solution for GPU memory management is that each task uses the native `cudaMallocManaged` function for GPU memory allocation, and delegates model transmission to CUDA unified memory. This solution incurs high overhead for DL applications because of two reasons. First, DL applications have large models and generate large amounts of intermediate results, which require a lot of GPU memory. Second, the native `cudaMalloc` function and CUDA unified memory are designed for general-purpose applications, and may incur unnecessary overhead for DL applications.

We exploit two characteristics of DL applications to minimize GPU memory management overhead. A DL task stores two important types of data in the GPU memory: the DNN model (including the model parameters), and the intermediate results. First, the amount of memory allocated to the DNN model is *fixed*, and does not change during task execution. An

inference task only uses the model for inference, and does not change the model itself. While a training task updates the model, it only updates the model parameters (i.e., the weights of the neural network), not the DNN structure, and the amount of memory needed to store them stays the same.

Second, the intermediate results change in a simple, regular pattern, which do not cause *memory fragmentation*. For an inference task, the intermediate results are the outputs of each layer, which are used by the next layer. After the next layer is computed, they are no longer needed and can be safely freed. A training task differs in that the intermediate results generated in the forward pass cannot be immediately freed, because they are also used by the backward pass to update the weights. However, the backward pass consumes the intermediate results in the reverse order as that the forward pass generates them, i.e., the intermediate results are first-in-last-out. The memory allocation and release can be handled by a simple stack-like mechanism, without causing memory fragmentation. The general-purpose GPU memory management does not consider these characteristics, and is too heavy-weight for DL applications that require fast task switching.

Minimize memory allocation overhead. Based on these two characteristics, we design a memory management mechanism tailored for DL applications. PipeSwitch uses a dedicated memory daemon to manage the GPU memory. To be compatible with the existing system and incur minimal changes, instead of replacing the GPU memory manager, the memory daemon uses `cudaMalloc` to obtain the GPU memory when the system starts, and then dynamically allocates the memory to the workers at runtime. This eliminates the overhead for each worker to use `cudaMalloc` to get a large amount of memory to store their models and intermediate results. The memory daemon only needs to pass memory pointers to the workers, which is light-weight. The daemon ensures that each time only one worker owns the GPU memory to guarantee memory isolation between workers. Each worker uses a memory pool to allocate the memory to store its model and intermediate results, and recycles the memory to the pool after the intermediate results are no longer needed.

The memory management of PipeSwitch extends that of PyTorch. It is designed and optimized for efficient GPU memory allocation between different tasks, while the memory management in PyTorch handles memory allocation for a task itself. PipeSwitch inserts GPU memory blocks to PyTorch GPU memory pool, and PyTorch creates tensors on them.

Minimize memory footprint and avoid extra memory copies. The server stores the DNN models in the host memory. Replicating the models in each worker incurs high memory footprint, and reduces the number of models a server can store, and consequently the types of tasks the server can execute. On the other hand, storing the models in a dedicate process has minimal memory footprint as each model is only stored once, but it incurs an extra memory copy from this process to a worker to start a task, which hurts the task switching time. We use unified memory management with the memory daemon to both achieve minimal memory footprint and eliminate extra memory copies. PipeSwitch stores the models in the memory daemon so that the server only needs to keep one copy of each model in the host memory. Because the memory daemon also manages the GPU memory, it directly transmits the model from the host memory to the GPU memory for task startup, which eliminates the extra memory copy from the memory daemon to the worker.

Minimize IPC overhead. After the model is transmitted to the GPU, the memory daemon needs to notify the worker and export the relevant GPU memory handlers to the worker, so that the worker can access the model to execute its task. This can be implemented by IPC APIs provided by GPUs, e.g., `cudaIpcOpenMemHandle` for NVIDIA GPUs. We have measured the performance of these IPC APIs and found that they incur high overhead (§6). The overhead is exacerbated by the pipeline because the pipeline needs to invoke the IPCs frequently to synchronize model transmission and task execution for *every pipeline group*, instead of invoking the IPC only once for the entire model transmission.

We leverage a property of DL applications to minimize the IPC overhead. The property is that the memory allocation process for a neural network model is *deterministic*. Specifically, given the same GPU memory region and the same model, as long as the memory daemon and the worker uses the same *order* to allocate memory for the model parameters, the memory pointers for the parameters would be the *same*. It is easy to keep the same order for the memory daemon and the worker because the neural network model is known and given, and the memory daemon only needs to use the same order to transmit the model as the worker would. As a result, the memory daemon can minimize the usage of expensive GPU IPCs. It only uses the GPU IPC once to initialize the worker, and then uses cheap CPU IPCs to notify the worker which pipeline group has been transmitted.

Pin memory. The OS would swap a memory page to disk if the page is inactive for a certain amount of time. GPUs require a page in the host memory to be pinned (or page-locked) in order to transmit the data in the page to the GPU memory. Otherwise, a temporary pinned page is created for the transmission. We pin the pages of the memory daemon to the host memory, to eliminate this overhead.

4.4 Active-Standby Worker Switching

PipeSwitch aims to provide fast task switching and ensure process-level isolation. Process-level isolation is desirable because it ensures that one task cannot read the memory of another task, and that the crashing of one task, e.g., because of a bug, does not affect other tasks or the entire system.

A naive solution is to use separate processes and start the new task after the current task is stopped. As we have profiled

	No Task Cleaning Overhead	No Task Initialization Overhead	Process-Level Isolation
Two Processes	×	×	√
One Process	×	√	×
Active-Standby	√	√	√

Table 2: Comparison of worker switching mechanisms.

in Table 1, such sequential execution incurs long delay due to old task cleaning and new task initialization.

Another possible solution is to let the current and new tasks share the same process with a warm CUDA context, so that the new task can reuse the GPU environment of the current task. This avoids the new task initialization, but it still has the overhead for the current task to clean its status. In addition, it does not provide process-level isolation between tasks.

We design an active and standby worker switching mechanism that hides the overhead of both task cleaning and task initialization, and also ensures process-level isolation. Similar to the naive solution, we use separate processes to achieve process-level isolation. PipeSwitch has an active worker and multiple standby workers. Each worker is a separate process, and initializes its own GPU environment (i.e., CUDA context) when it is first created. This eliminates the GPU environment initialization overhead when a new task is assigned to a worker. When a current task is stopped, a major job is to clear asynchronous CUDA functions queued on the GPU. We insert synchronization points into training tasks, so the number of queued functions are limited and can be quickly cleared. Synchronization points are not needed for inference tasks as they are short and not preempted. Another job is to free its GPU memory. An important property of the cleaning procedure is that it does not modify the content of the memory, but only cleans the metadata, i.e., GPU memory pointers. As the GPU memory is managed by PipeSwitch, the cleaning procedure deletes the pointers pointing to the tensor data rather than freeing the actual data. Therefore, it is safe for the new task to transmit its model to the GPU memory at the same time. In other words, we can parallelize the task cleaning of the current task and the pipelined model transmission of the new task, to hide the task cleaning overhead. This choice is optimized for performance, and is not a problem for a trusted environment. It is possible that a latter process can read the memory data of a previous process. If this is a concern, an additional zero-out operation can be added. GPU has high memory bandwidth (e.g., 900GB/s for V100). It would incur sub-millisecond overhead for zeroing-out most models like ResNet-152 (around 240MB). On the other hand, for a trusted environment, it is unnecessary to release all allocated memory for the preempted process if the new process does not require entire GPU memory, and this could be achieved by some simple coordination. Table 2 summarizes the differences between these three solutions.

In summary, to switch workers, the controller signals the current active worker to stop, deletes the GPU memory allocated to it, and allocates the GPU memory to the new active worker. The controller ensures only one active worker to guarantee exclusive occupation of the GPU.

There is a trade-off between the number of standby workers and their GPU memory consumption. On one hand, task cleaning takes time. If a new task arrives before a standby worker finishes cleaning a previous task, the new task needs to wait, which increases its startup time. On the other hand, it is possible to have many standby workers so that there is always at least one idle standby worker. However, every standby worker needs to maintain its own CUDA context, which consumes a few hundred MB GPU memory. Our experience is that two standby workers are sufficient to ensure at least one idle worker, which eliminates the waiting time and has moderate GPU memory consumption.

4.5 Discussion

PipeSwitch is focused on single-GPU tasks for training and inference. For inference tasks, strict SLOs require requests to be handled in small batches for low latency, so it is common to execute an inference task with a single GPU [18]. Multi-GPU inference tasks can be supported by performing PipeSwitch on each GPU with *transactions*. A transaction here means a model is switched in or out on all of its GPUs to enable or disable inference on this model.

For training tasks, PipeSwitch supports single-GPU training and asynchronous multi-GPU training for data parallel strategies, as preempting one GPU does not affect other GPUs. However, it does not work out of the box with synchronous multi-GPU training. We have analyzed a production GPU training trace from Microsoft [19, 20]. Among 111,883 tasks in this trace, 96,662 tasks (or 86% of all the tasks) are single-GPU training tasks. Thus, a significant fraction of tasks in real-world workloads currently use a single GPU, and PipeSwitch is applicable to them out of the box. However, these jobs only account for 18% of total GPU hours and we expect the share of multi-GPU jobs to increase in the future. One way to seamlessly use PipeSwitch for synchronous multi-GPU training is to use elastic synchronous training, which allows the dynamic changing of the number of GPUs used for training. Unfortunately, current training frameworks do not have mature support of elastic training. This remains an active research topic and is orthogonal to PipeSwitch.

5 Implementation

We have implemented a system prototype for PipeSwitch with ∼3600 lines of code in C++ and Python, and we have integrated it with PyTorch [21].

PyTorch Plugins. We add C++ and Python functions to the GPU memory management module of PyTorch. To share GPU memory between the controller and the workers, we add functions for allocating GPU memory, sharing the GPU

memory to workers through CUDA IPC API, and getting the shared GPU memory. We also add functions which insert the received GPU memory into PyTorch GPU memory pool for a specific CUDA stream or clear the GPU memory from the pool. Note that the shared GPU memory can be inserted into the PyTorch GPU memory pool for multiple times for different CUDA streams, and the controller guarantees that only one of these CUDA streams is active.

Controller and memory daemon. The controller process consists of a TCP thread and a scheduler thread. For better performance, the scheduler and the memory daemon are implemented together. The TCP thread accepts task through TCP from clients, and sends the task to the scheduler thread. The scheduler thread allocates and shares the GPU memory with workers, activates or deactivates workers, sends the task to a worker, and transfers parameters for the corresponding model to the GPU memory. Before starting a task, the user should register the model in the scheduler to notify the controller to load the model from the disk to the CPU memory. When the controller schedules a task, it determines whether to switch to another worker. There is no need for context switching if the application is already loaded in the GPU. If a new model should be loaded to the GPU, the controller will notify the current active worker to stop, and transfers the parameters of the new model to the GPU after receiving the current active worker's reply. Parameters are transmitted to the GPU memory in groups in a pipeline. After each group is transferred, the controller notifies the worker to start computing the corresponding layers.

Worker. The worker process consists of two threads. The termination thread waits for the termination signal from the controller, and notifies the main thread. The main thread manages the DNN models and performs the computation for inference or training. Similar to the controller, the worker also requires the user to register the model before starting a task, so the worker can load the models and add the hooks to wait for parameter transmission or terminate on notification. Note that the worker only loads the model structures, which is small, not the model parameters. The parameters are only stored once in the memory daemon for minimal memory footprint. When the models are loaded, they are attached to different CUDA streams, and their parameters are assigned to locations in the shared GPU memory. Different models might use the same GPU memory location, but the value is not valid until the controller transfers the corresponding parameters to these locations. After loading the models, the worker waits for the scheduler to transfer required parameters for DNN models, and performs inference or training.

6 Evaluation

In this section, we first use end-to-end experiments to demonstrate the benefits of PipeSwitch, and then show the effectiveness of the design choices on each component.

Setup. All experiments are conducted on AWS. We use two EC2 instance types. One is p3.2xlarge, which is configured with 8 vCPUs (Intel Xeon E5-2686 v4), 1 GPU (NVIDIA V100 with 16 GB GPU memory), PCIe 3.0 $\times$16, and 61 GB memory. The other is g4dn.2xlarge, which is configured with 8 vCPUs (Intel Platinum 8259CL), 1 GPU (NVIDIA T4 with 16 GB GPU memory), PCIe 3.0 $\times$8, and 32 GB memory. The software environment includes PyTorch-1.3.0, torchvision-0.4.2, scipy-1.3.2, and CUDA-10.1. We use PyTorch with our plugins for all mechanisms in comparison for consistency, which provides better results for stop-and-start than native PyTorch from Python-PyPI used in Table 1.

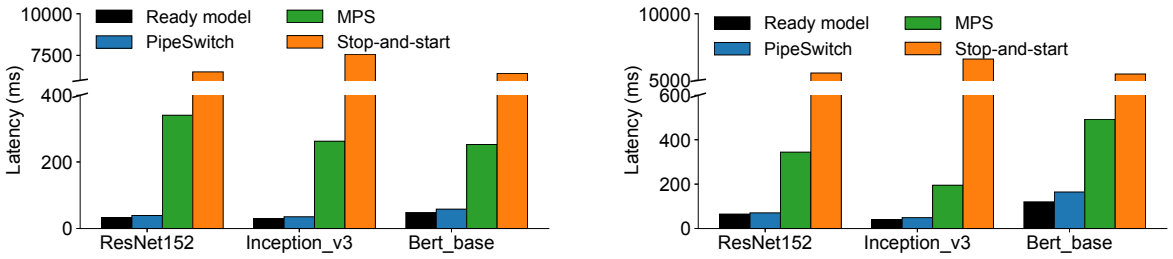
Workloads. The models include ResNet152 [17], Inception_v3 [22] and Bert_base [23], which are standard benchmarks for evaluating DL systems. We use representative configurations for each model. The experiments cover both training and inference. We use single-GPU inference and training tasks as discussed in §4.5. Training tasks periodically checkpoint their models to the host memory, and restart from the latest checkpoint after preemption. The checkpointing frequency of training tasks is set according to the scheduling cycle to minimize checkpointing overhead. The default batch size for training is 32, and that for inference is 8.

Metrics. We use throughput and latency as evaluation metrics. Each number is reported with the *average* of 100 runs. For Figure 6(b), we additionally report the minimum and maximum latencies using the error bar, because the latency of the first batch and those of later batches in one scheduling cycle can differ significantly due to switching overhead.

6.1 End-to-End Experiments

Minimizing end-to-end overhead. In this experiment, a client sends an inference task to a GPU server, and the GPU server preempts the training task to execute the inference task and sends a reply back to the client. We measure the the end-to-end latency experienced by the client. We compare the following mechanisms.

- **Ready model.** There is no training task. The process with the required model is already loaded in the GPU. This solution provides the lower bound, which is the lowest latency we can achieve for an inference task.
- **Stop-and-start.** It stops the training task in the GPU, and then starts the inference task. This solution is used by existing systems like Gandiva [24] for task switching, which reported similar *second-scale* overhead.
- **NVIDIA MPS.** This is the multi-process support from NVIDIA which allows the inference process to share the GPU with the training process. We initialize separate processes in advance. The training task occupies the entire GPU memory and does not stop when inference tasks come. CUDA unified memory is used for memory swapping.
- **PipeSwitch**. This is the proposed system. The properties are described in §4.



(a) p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16).

(b) g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8).

Figure 5: Total latency experienced by the client for different mechanisms.

	p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16)			g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8)		
	ResNet152	Inception_v3	Bert_base	ResNet152	Inception_v3	Bert_base
Stop-and-start	6475.40 ms	7536.07 ms	6371.32 ms	5486.74 ms	6558.76 ms	5355.95 ms
NVIDIA MPS	307.02 ms	232.25 ms	204.52 ms	259.20 ms	193.05 ms	338.25 ms
PipeSwitch	6.01 ms	5.40 ms	10.27 ms	5.57 ms	7.66 ms	34.56 ms

Table 3: Total overhead, i.e., the difference on total latency between different mechanisms and ready model.

	ResNet152	**Inception_v3**	**Bert_base**
p3.2xlarge	3.62 ms	4.82 ms	3.62 ms
g4dn.2xlarge	2.53 ms	5.49 ms	6.57 ms

Table 4: The startup overhead for PipeSwitch to start computing the first layer.

Salus [7] is not directly comparable because it requires the models to be preloaded to the GPU, and has several limitations described in §2.2. Its performance is similar to the ready model when the model is preloaded, and is similar to NVIDIA MPS when the model is in the host memory. Figure 5 shows the latency experienced by the client, and Table 3 shows the total overhead. The total overhead is the difference between the latency of a mechanism and that of the *ready model*. It is obvious that stop-and-start performs the worst, which takes several seconds. The main source of the overhead is CUDA context initialization and first-time library loading operations in PyTorch. NVIDIA MPS has lower overhead compared to stop-and-start, but still incurs several hundred milliseconds overhead, which prevents MPS from meeting strict SLOs. One source of the overhead is the contentions both on the computation and memory of the GPU, as the training task do not stop when an inference task comes. Another source is GPU memory swapping. PipeSwitch performs the best and is close to the lower bound. The overhead of PipeSwitch for most configurations is up to 10ms, except for BERT on T4, which is due to the large model size and the smaller PCIe bandwidth on T4 than that on V100. Since it also takes longer (120ms) to compute BERT on T4 even with the ready model, the relative overhead is acceptable.

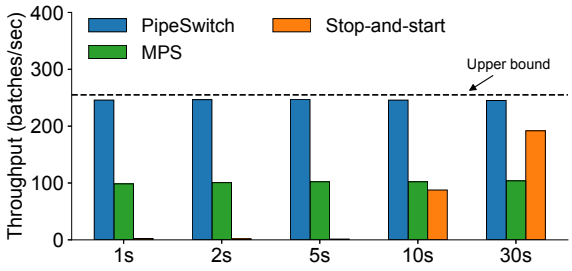
We also show the task startup overhead for PipeSwitch in Table 4, which is the difference between the time for PipeSwitch to start computing the first layer and that for the ready model to start computing. The startup overhead of PipeSwitch is only a few milliseconds.

	ResNet152	**Inception_v3**	**Bert_base**
# of Layers	464	189	139
Algorithm 1	1.33 s	0.18 s	0.34 s
Only Pruning 1	2.09 s	0.30 s	0.88 s
Only Pruning 2	3.44 h	5.07 s	> 24 h
No Pruning	> 24 h	> 24 h	> 24 h

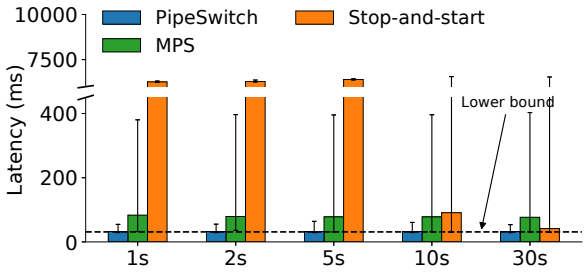
Table 5: Effectiveness of two pruning techniques.

Enabling fine-grained scheduling cycles. In this experiment, we compare throughput and end-to-end latency of different mechanisms under different scheduling cycles. We use ResNet152 for both training and inference on eight p3.2xlarge instances, and switch between these two tasks after each scheduling cycle. Figure 6(a) shows the inference throughput. The dashed line is the upper bound, which is the throughput of the ready model assuming no task switching. The throughput of stop-and-start is nearly zero for scheduling cycles smaller than 10 s, because it takes several seconds for task switching. MPS keeps poor throughput around 100 batches per second. We define GPU utilization as the ratio to the upper bound. PipeSwitch has high throughput close to the upper bound, achieving near 100% GPU utilization.

Figure 6(b) shows the average latency of the inference tasks. The dashed line is the lower bound, which is the average latency of the ready model assuming no task switching. The error bar indicates the minimum and maximum latency. Stop-and-start has poor latency because the first batch has several seconds overhead. MPS has about 80 ms average latency, and has several hundred milliseconds latency for the first batch.



(a) Throughput (eight p3.2xlarge instances).



(b) Latency.

Figure 6: Throughput and latency under different scheduling cycles for ResNet on p3.2xlarge.

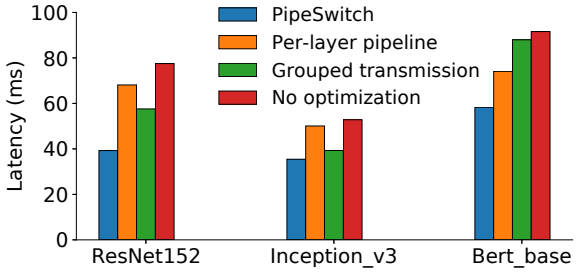
PipeSwitch incurs only a few milliseconds overhead for task switching, and achieves low latency close to the lower bound.

6.2 Pipelined Model Transmission

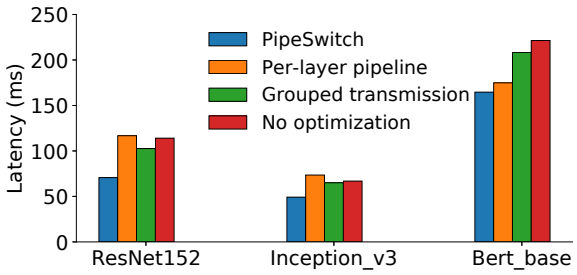
To evaluate the effectiveness of pipelined model transmission, we keep all other components of PipeSwitch the same, and compare the following mechanisms discussed in §4.2.

- **No optimization.** It transmits the model layer by layer (with many PCIe calls), and then executes the task.
- **Grouped transmission.** It groups the entire model in one transmission, and then executes the task.
- **Per-layer pipeline.** It transits model parameters layer by layer. Computation starts, once parameters are transmitted.
- **PipeSwitch**. It is the pipelining mechanism with optimal model-aware grouping in PipeSwitch.

Figure 7 shows the total time measured by the client for an inference task to preempt a training task and finish its inference. No optimization performs the worst in most cases. Grouped transmission improves no optimization by combining the layers of the model into one big tensor and transmitting it in one group. Per-layer pipeline overlaps transmission and computation at the granularity of layer. But because it has PCIe overhead and synchronization overhead for every layer, for the models with many layers but relatively light computation such as ResNet152 and Inception, it can perform worse than grouped transmission and sometimes even no pipeline. PipeSwitch uses model-aware grouping and achieves the best trade-off between pipeline overhead and efficiency. It reduces the total time by up to 38.2 ms compared to other solutions.



(a) p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16).



(b) g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8).

Figure 7: Effectiveness of pipelined model transmission.

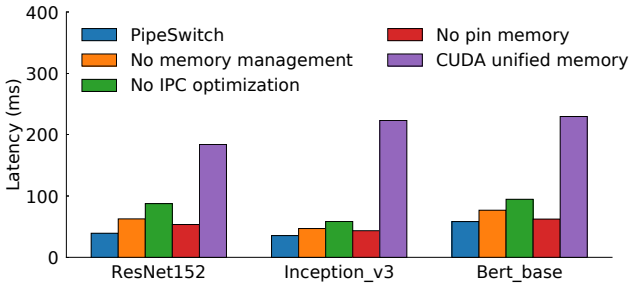
Note that this reduction is significant, especially considering that it is evaluated when the optimizations on memory management and worker switching have already been applied. We would like to emphasize that to meet strict SLOs, it is important to reduce all overheads for task switching, not only the most significant one.

Table 5 shows the running time of Algorithm 1, as well as the effects of the two pruning techniques mentioned in § 4.2. Note that the number of layers includes both weighted and unweighted layers, as both contribute to the computation time. We measure the parameter size and running time for each layer in advance. Algorithm 1 takes only several seconds to compute an optimal grouping strategy, even for ResNet152 which has hundreds of layers. On the contrary, no pruning does not finish for all three models after running for 24 hours.

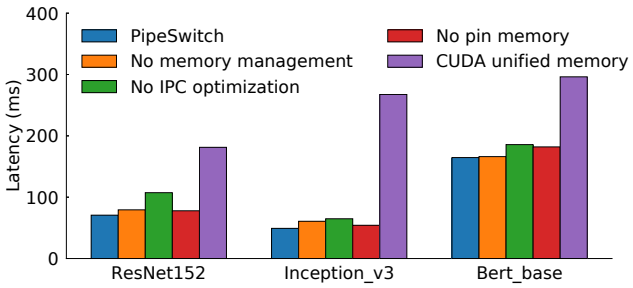
6.3 Unified Memory Management

To evaluate the effectiveness of unified memory management. we keep all other components of PipeSwitch the same, and compare the following five mechanisms discussed in §4.3.

- **No unified memory management.** Each worker uses `cudaMalloc` to allocate GPU memory, and transmits the model to GPU by its own.
- **No IPC optimization.** The memory daemon handles GPU memory allocation and model transmission, but creates and sends GPU memory handlers to workers. To compare, PipeSwitch simply sends an 64-bit integer offset for the shared GPU memory to workers.
- **No pin memory.** It has all optimizations on unified memory management except that the pages of the memory daemon are not pinned to the main memory.

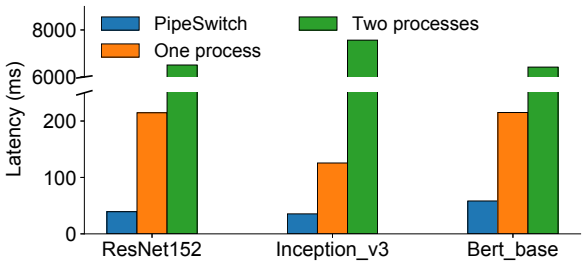


(a) p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16).

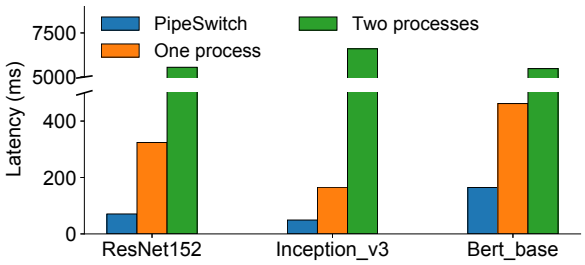


(b) g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8).

Figure 8: Effectiveness of unified memory management.



(a) p3.2xlarge (NVIDIA V100, PCIe 3.0 ×16).



(b) g4dn.2xlarge (NVIDIA T4, PCIe 3.0 ×8).

Figure 9: Effectiveness of active-standby switching.

- **CUDA unified memory.** Each worker allocates GPU memory with `cudaMallocManaged`, and CUDA automatically transmits the model to GPU when needed.
- **PipeSwitch**. It is the unified memory management mechanism used by PipeSwitch.

Figure 8 shows the total time measured by the client. First, compared to no unified memory management, PipeSwitch saves 2–23 ms by eliminating the memory allocation overhead with the memory daemon. It is also important to note that no unified memory management requires each worker to keep a copy for each DNN model, which increases the memory footprint. Second, IPC optimization is important, which reduces the latency by 16–48 ms. Without IPC optimization, the latency is even higher than no unified memory management. Third, pinning the pages to the host memory can reduce the latency with a few milliseconds. Finally, CUDA unified memory is not optimized for DL applications, and introduces more than one hundred milliseconds overhead than PipeSwitch. Overall, this experiment demonstrates that all the optimizations on memory management are effective.

6.4 Active-Standby Worker Switching

To evaluate the effectiveness of active-standby worker switching, we keep all other components of PipeSwitch the same, and compare the following mechanisms discussed in §4.4.

- **Two processes.** The process of the old task cleans the GPU environment, and then another process is created and initialized for the new task.
- **One process.** The process cleans the GPU environment for the old task, and reuses the environment for the new task.
- **PipeSwitch** It is the active-standby worker switching mechanism used by PipeSwitch.

Figure 9 shows the results. Two processes perform the worst as it stops the training task and initializes a new process for the new task. The new process needs to create a new CUDA environment, which dominates the total time. One process reuses the CUDA environment, but still pays the overhead to clean the environment. PipeSwitch uses an active-standby worker switching mechanism to parallelize old task cleaning and new task initialization, and incurs minimal overhead. It reduces the latency by 116–307 ms compared to one process, and 5–7 s compared to two processes.

7 Related Work

Many frameworks have been developed for deep learning, such as TensorFlow [25], PyTorch [21] and MXNet [26]. Several algorithms and systems have been designed for executing and scheduling deep learning tasks on clusters, including both training and inference tasks [3, 10, 24, 27–32]. These scheduling solutions are orthogonal and complementary to PipeSwitch. They focus on what scheduling decisions to make, while PipeSwitch focuses on how to realize a scheduling decision. Importantly, PipeSwitch enables the scheduler to change the resource allocation more often with millisecond-scale task switching. Many techniques and systems have been proposed to optimize communication and improve distributed training [8, 9, 15, 33–42]. The most relevant ones are PipeDream [8], ByteScheduler [9] and Poseidon [40]. They use inter-batch pipelining for training of the same task, while PipeSwitch introduces intra-batch pipelining to fast start both

training and inference tasks and enables fast switching across tasks. Other works like vDNN [43] and SwapAdvisor [44] also have GPU memory management module, but they focus on memory management for a single training task of large models, which are not directly comparable to PipeSwitch.

Cluster managers [45–48] typically allocate GPUs to VMs or containers at device granularity. Several solutions have been proposed to share a GPU at application granularity using techniques like library interception [6, 49–53]. They are general-purpose and focus on sharing only a few kernels. As such, they are not suitable for deep learning applications that typically require hundreds of kernels. NVIDIA MPS [6] provides official support for sharing a GPU between multiple processes. It is also not specially designed for deep learning and thus cannot meet strict SLOs of inference tasks as shown in §6. There are many efforts on GPU optimization to improve the performance of running a single task, such as tensor fusion and kernel-level concurrency and scheduling [54–58]. These solutions are complementary to PipeSwitch.

8 Conclusion

We present PipeSwitch, a system that enables GPU-efficient fine-grained time-sharing for multiple DL applications. We introduce pipelined context switching to minimize task switching overhead on GPUs for DL applications. Pipelined context switching includes three key techniques, which are pipelined model transmission, unified memory management and active-standby worker switching. With these techniques, PipeSwitch is able to achieve millisecond-scale task switching time, and enables DL applications on time-sharing GPUs to meet strict SLOs. We demonstrate the performance of PipeSwitch with experiments on a variety of DNN models and GPU cards. PipeSwitch can significantly increase GPU utilization and improve the agility of DL applications.

Acknowledgments. We thank our shepherd Madan Musuvathi and the anonymous reviewers for their valuable feedback. Zhihao Bai, Zhen Zhang and Xin Jin were supported in part by an AWS Machine Learning Research Award.

References

[1] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*, 2015.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, 2013.

[3] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis," in *ACM SOSP*, 2019.

[4] "CUDA Unified Memory." https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

[5] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *USENIX NSDI*, 2019.

[6] "CUDA Multi-Process Service." https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[7] P. Yu and M. Chowdhury, "Salus: Fine-grained GPU sharing primitives for deep learning applications," in *Conference on Machine Learning and Systems*, 2020.

[8] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: generalized pipeline parallelism for DNN training," in *ACM SOSP*, 2019.

[9] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *ACM SOSP*, 2019.

[10] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *USENIX NSDI*, 2019.

[11] M. Jeon, S. Venkataraman, A. Phanishayee, u. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *USENIX ATC*, 2019.

[12] "Amazon Web Services." https://aws.amazon.com/.

[13] "Microsoft Azure." https://azure.microsoft.com/.

[14] "Google Cloud Platform." https://cloud.google.com/.

[15] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[16] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *USENIX OSDI*, 2014.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[18] "Nvidia data center deep learning product performance." https://developer.nvidia.com/deep-learning-performance-training-inference.

[19] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *USENIX ATC*, 2019.

[20] "Philly traces." https://github.com/msr-fiddle/philly-traces.

[21] "PyTorch." https://pytorch.org/.

[22] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019.

[24] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *USENIX OSDI*, 2018.

[25] "TensorFlow." https://www.tensorflow.org/.

[26] "MXNet." https://mxnet.apache.org/.

[27] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *ACM Symposium on Cloud Computing*, 2017.

[28] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *EuroSys*, 2018.

[29] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient GPU cluster scheduling," in *USENIX NSDI*, 2020.

[30] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov, "HyperSched: Dynamic resource reallocation for model development on a deadline," in *ACM Symposium on Cloud Computing*, 2019.

[31] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An optimizing compiler for fully-homomorphic neural-network inferencing," in *ACM Conference on Programming Language Design and Implementation*, 2019.

[32] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *USENIX OSDI*, 2018.

[33] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019.

[34] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica, "Blink: Fast and generic collectives for distributed ML," in *Conference on Machine Learning and Systems*, 2020.

[35] "NVIDIA Collective Communications Library (NCCL)." https://developer.nvidia.com/nccl.

[36] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over infiniband," *Int. J. Parallel Program.*, vol. 32, 2004.

[37] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *Advances in Neural Information Processing Systems*, 2013.

[38] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda, "Optimized broadcast for deep learning workloads on dense-GPU infiniband clusters: MPI or NCCL?," in *Proceedings of the 25th European MPI Users' Group Meeting*, 2018.

[39] J. Daily, A. Vishnu, C. Siegel, T. Warfel, and V. Amatya, "GossipGraD: Scalable deep learning using gossip communication based asynchronous gradient descent," *CoRR*, vol. abs/1803.05880, 2018.

[40] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *USENIX ATC*, 2017.

[41] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, "Is network the bottleneck of distributed training?," in *ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, August 2020.

[42] Y. Chen, Z. Liu, B. Ren, and X. Jin, "On efficient constructions of checkpoints," in *International Conference on Machine Learning (ICML)*, July 2020.

[43] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[44] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *ACM ASPLOS*, 2020.

[45] "Kubernetes." https://kubernetes.io/.

[46] "NVIDIA Container Runtime for Docker." https://github.com/NVIDIA/nvidia-docker.

[47] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.," in *USENIX NSDI*, 2011.

[48] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *ACM Symposium on Cloud Computing*, 2013.

[49] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *European Conference on Parallel Processing*, 2010.

[50] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, 2009.

[51] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing & Simulation*, 2010.

[52] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th international symposium on High performance distributed computing*, 2011.

[53] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, 2011.

[54] "TensorFlow XLA." https://www.tensorflow.org/xla/.

[55] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[56] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, 2012.

[57] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," *ACM SIGARCH Computer Architecture News*, vol. 41, 2013.

[58] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: optimizing deep learning computation with automatic generation of graph substitutions," in *ACM SOSP*, 2019.

HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees

Hanyu Zhao [1,3*], Zhenhua Han [2,3*], Zhi Yang [1], Quanlu Zhang [3], Fan Yang [3], Lidong Zhou [3], Mao Yang [3], Francis C.M. Lau [2], Yuqi Wang [3], Yifan Xiong [3], Bin Wang [3]
[1] *Peking University,* [2] *The University of Hong Kong,* [3] *Microsoft*

Abstract

Deep learning training on a shared GPU cluster is becoming a common practice. However, we observe severe sharing anomaly in production multi-tenant clusters where jobs in some tenants experience worse queuing delay than they would have in a private cluster with their allocated shares of GPUs. This is because tenants use *quota*, the number of GPUs, to reserve resources, whereas deep learning jobs often use GPUs with a desirable *GPU affinity*, which quota cannot guarantee.

HiveD is the first framework to share a GPU cluster *safely*, so that such anomaly would never happen by design. In HiveD, each tenant reserves resources through a Virtual Private Cluster (VC), defined in terms of multi-level *cell* structures corresponding to different levels of GPU affinity in a cluster. This design allows HiveD to incorporate any existing schedulers within each VC to achieve their respective design goals while sharing the cluster safely.

HiveD develops an elegant *buddy cell allocation* algorithm to ensure *sharing safety* by efficiently managing the dynamic binding of cells from VCs to those in a physical cluster. A straightforward extension of buddy cell allocation can further support low-priority jobs to scavenge the unused GPU resources to improve cluster utilization.

With a combination of real deployment and trace-driven simulation, we show that: (i) sharing anomaly exists in three state-of-the-art deep learning schedulers, incurring extra queuing delay of up to 1,000 minutes; (ii) HiveD can incorporate these schedulers and eliminate the sharing anomaly in all of them, achieving separation of concerns that allows the schedulers to focus on their own scheduling goals without violating sharing safety.

1 Introduction

Deep learning training is becoming a major computing workload on a GPU cluster. It is a common practice for an organization to train deep learning models in a multi-tenant GPU cluster, where each tenant reserves resources using a quota that consists of the number of GPUs and other associated resources such as CPU and memory [52].

Surprisingly, in a production multi-tenant GPU cluster, we have observed unexpected anomalies where a tenant's deep learning training jobs wait significantly longer for GPUs than they would do in a private cluster whose size equals to the tenant's quota. This is because the current resource reservation mechanism is based on *quota*, i.e., the number of GPUs. Quota cannot capture the GPU *affinity* requirement of training jobs: e.g., an 8-GPU job on one node usually runs significantly faster than on eight nodes [41, 52, 86]. Quota cannot guarantee a tenant's GPU affinity like the tenant's private cluster does. As a result, multi-GPU jobs often have to wait in a queue or run at a relaxed affinity, both resulting in worse performance (longer queuing delay or slower training speed).

In this paper, we present HiveD, a resource reservation framework to share a GPU cluster for deep learning training that guarantees *sharing safety* by completely eliminating sharing anomalies. Instead of using quota, HiveD presents each tenant a virtual private cluster (abbreviated as VC) defined by a new abstraction: *cell*. Cell uses a multi-level structure to capture the different levels of affinity that a group of GPUs could satisfy. Those cell structures naturally form a hierarchy in a typical GPU cluster; e.g., from a single GPU, to GPUs attached to a PCIe switch, to GPUs connected to a CPU socket, to GPUs in a node, to GPUs in a rack, and so on.

With cell, HiveD virtualizes a physical GPU cluster as a VC for each tenant, where the VC preserves the necessary affinity structure in a physical cluster. This allows any state-of-the-art deep learning scheduler to make scheduling decisions within the boundary defined by the VC, without affecting the affinity requirement from other VCs, hence ensuring sharing safety. In this way, HiveD achieves the separation of concerns [47]: It focuses on the resource reservation mechanism and leaves other resource allocation goals to VC schedulers (e.g., cluster utilization and job completion time).

HiveD develops an elegant and efficient *buddy cell allocation* algorithm to bind cells from a VC to a physical cluster. Buddy cell allocation advocates dynamic cell binding over static binding for flexibility. It dynamically creates and releases the binding of cells in a VC to GPUs in the physical cluster, while providing *proven* sharing safety despite unpredictable workloads. Moreover, the algorithm can be naturally extended to support preemptible low-priority jobs to scavenge unused cells opportunistically to improve overall utilization. Combined, HiveD achieves the best of both a private cluster (for guaranteed availability of cells independent of other tenants) and a shared cluster (for improved utilization and access to more resources when other tenants are not using them).

We evaluate HiveD using experiments on a 96-GPU real

*Equal contribution.

cluster and trace-driven simulations. The evaluation shows that (i) sharing anomaly exists in all the evaluated state-of-the-art deep learning schedulers [41,52,86]; (ii) HiveD eliminates all sharing anomalies, decreases excessive queuing delay from 1,000 minutes to zero, while preserving these schedulers' design goals; (iii) HiveD guarantees sharing safety regardless of cluster loads, whereas a quota-based cluster can result in 7× excessive queuing delay for a tenant under a high load.

We have open-sourced HiveD [17], and integrated it in OpenPAI [20], a Kubernetes-based deep learning training platform. It has been deployed in multiple GPU clusters serving research and production workloads at scale, including a cluster of 800 GPUs where HiveD has been up and running reliably for more than 12 months (as of Nov. 2020).

In summary, this paper makes the following contributions:

- We are the first to observe and identify sharing anomaly in production multi-tenant GPU clusters for deep learning training.
- We define the notion of sharing safety against the anomaly and propose a new resource abstraction, i.e., multi-level cells, to model virtual private clusters.
- We develop an elegant and efficient buddy cell allocation algorithm to manage cells with proven sharing safety, and to support low-priority jobs.
- We perform extensive evaluations both on a real cluster and through simulation, driven by a production trace, to show that HiveD achieves the design goals in terms of sharing safety, queuing delay, and utilization.

2 Background and Motivation

The current approach of managing a multi-tenant GPU cluster. In large corporations, a large-scale GPU cluster is usually shared by multiple business teams, each being a tenant contributing their resources (budget or hardware). The tenants share the GPU cluster in a way similar to sharing a CPU cluster [1, 52]: Each tenant is assigned a number of tokens as its *quota*. Each token corresponds to the right to use a GPU along with other types of resource. The quota denotes an expectation that the tenant can access "at least" the share of resources it contributes.

To improve training speed in the cluster, a user usually specifies a GPU *affinity requirement* for a deep learning job [52, 86]. For example, it is often desirable for a 64-GPU job to run in the 8×8 affinity, i.e., to run the job on 8 nodes each with 8 GPUs, instead of 64×1, i.e., 64 nodes each using 1 GPU. Given the affinity requirements, the resource manager will satisfy them in a guaranteed (hard) or best-effort (soft) manner. If there is no placement satisfying a job's affinity requirement, the job will wait in the queue if it has a hard affinity requirement or will be scheduled with relaxed affinity if the requirement is soft (e.g., 64×1 as opposed to 8×8).

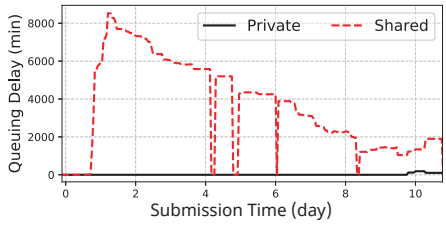


Figure 1: Sharing anomaly: a tenant suffers longer queuing delay in a shared cluster than in its own private cluster.

Sharing anomaly. In a production GPU cluster described in [52], we observe an anomaly from user complaints: a tenant is assigned a quota of 64 GPUs but reports that it cannot run a single (and the only) 8×8 deep learning job. Such anomaly arises because the tenant's assigned affinity has been fragmented, not by its own job(s) but by jobs from other tenants. Even though the tenant has enough GPU quota, the 64-GPU job has to wait in a queue or execute with degraded performance with relaxed affinity. The promise to the tenant that it can access at least its share of resource is broken.

Sharing anomaly appears similar to external fragmentation in memory management [54], if we liken a tenant to a program. The important difference however is that, in a shared GPU cluster, tenants expect their resource shares to be guaranteed. In the above real-world example, the fragmentation is due to other tenants, and the suffering tenant can hardly do anything except to complain to the cluster operator. Sharing anomaly can easily happen when jobs with lower affinity requirement (e.g., single-GPU jobs) from a tenant add to the fragmentation of global resources (due to varying job arrival and completion times), making jobs with higher affinity requirement (e.g., 8×8-GPU jobs) from other tenant(s) not able to run, even with sufficient quota. Apparently, quota can reserve only the quantity of resources, but not the affinity of resources. Hence it cannot automatically get around the external fragmentation across tenants. We call this phenomenon "sharing anomaly" because the sharing of a tenant's resource impacts the tenant negatively. Therefore, in the above case, rather than sharing with others, the wised up tenant would prefer to run a private cluster with eight 8-GPU nodes to adhere to its 8×8 GPU affinity with zero queuing delay.

A multi-tenant cluster is said to suffer from *sharing anomaly* if a tenant's sequence of GPU requests (possibly with affinity requirement) cannot be satisfied in this shared cluster; whereas it can be satisfied in a private cluster whose size equals to the tenant's quota. Figure 1 highlights how severe sharing anomaly could become, selected from a trace-driven simulation in a setup similar to [52] (more details in §5). The figure shows the job queuing anomaly of one tenant in a shared cluster when jobs have hard affinity requirement. In the 10-day submission window (denoted as X-axis), the

tenant's average job queuing delay (denoted as Y-axis) in the shared cluster is significantly higher than that in its own private cluster.[1] In particular, the jobs submitted around Day 1 have to stay in the queue for more than 8,000 minutes (5 days) while they have zero queuing delay in the private cluster! Moreover, tenants having reserved large resources tend to suffer the most. Consequently, we have witnessed important corporate users reverting to private clusters, after experiencing high queuing delay brought by severe sharing anomalies.

One approach to reducing sharing anomaly is to devise a scheduling policy to minimize global resource fragmentation. This makes the design of a deep learning scheduler even more complex, which already has to manage sophisticated multi-objective optimizations. For example, minimizing global fragmentation may decrease job performance due to increased inter-job interference [86]. Therefore, we propose to separate the concern of sharing anomaly from other resource allocation objectives [47]. Instead of developing a complicated scheduler that achieves all possible goals, we design HiveD, a resource reservation framework that focuses on eliminating sharing anomaly, and provides a clean interface to incorporate any state-of-the-art deep learning schedulers to address concerns like cluster utilization [86], job completion time [41, 66], and fairness [29, 60].

3 HiveD Design

3.1 System Overview

HiveD proposes to guarantee *sharing safety* (i.e., eliminating sharing anomaly as described in §2) as a prerequisite of sharing a GPU cluster. Specifically, if a sequence of GPU requests with affinity requirements can be satisfied in a private cluster, it should be satisfied in the corresponding virtual private cluster and the shared physical cluster.

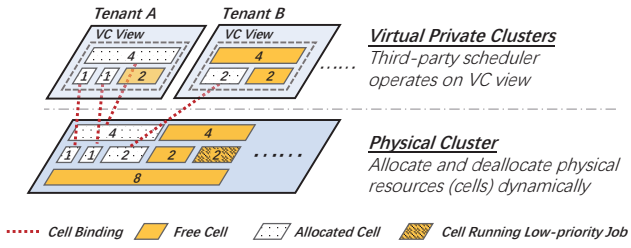


Figure 2: System architecture: a two-layer design.

Figure 2 illustrates the overall system architecture. HiveD's abstraction of GPU resources is divided into two layers, i.e., the layer of Virtual Private Clusters (VCs) and the layer of physical cluster. HiveD presents each tenant a VC. Each VC is pre-assigned a set of *cell*s, a novel resource abstraction that captures not only quota, but also the affinity structure of GPUs (the number inside each cell in the figure shows the number of affinitized GPUs of the cell). The cells assigned to a VC form a VC view with the GPU affinity structure identical to that of the corresponding private cluster. Any third-party scheduler can be incorporated to work on the VC view to achieve a certain goal of resource allocation [41, 52, 60, 86]. Moreover, HiveD ensures that any scheduling decision is constrained within the boundary defined by the VC view, as if happening on its private cluster, thus guaranteeing sharing safety.

Cells in a VC are logical. When a job uses a GPU in a logical cell, e.g., one GPU in the 4-GPU cell in the VC view of Tenant A in Figure 2, the logical cell will be bound to a physical cell allocated from the physical cluster, denoted at the bottom of Figure 2. If none of the GPUs is in use, the logical cell will be unbound from the physical cluster. To improve utilization, preemptible low-priority jobs can scavenge idle GPUs opportunistically. Such dynamic binding is more flexible than static binding: a dynamic binding can avoid a physical cell whose hardware is failing; it can avoid cells used by low-priority jobs to reduce preemptions; it can also pack the cells to minimize the fragmentation of GPU affinity.

To achieve this, HiveD adopts *buddy cell allocation*, an efficient and elegant algorithm, to handle the dynamic binding. A key challenge of dynamic binding is to guarantee the safety property in response to dynamic workloads, that is, jobs arrive unpredictably and request varying levels of cells. Buddy cell allocation algorithm is proven to ensure sharing safety: any legitimate cell request within a VC is guaranteed to be satisfied. The algorithm can also support low-priority jobs. Figure 2 shows a possible cell allocation, where cells in a physical cluster are bound to those defined in two VCs, and also to a low-priority job.

In §3.2, we explain the details of cells and show how a VC can be defined by cells. And in §3.3, we introduce the buddy cell allocation algorithm, prove its sharing safety guarantee, and extend it to support low-priority jobs.

3.2 Virtual Private Cluster with Cells

To model a (private) GPU cluster, HiveD defines a *hierarchy of multi-level cell structures*. A *cell* at a certain level is the corresponding collection of affinitized GPUs with their interconnection topology. Each virtual private cluster (VC) is then defined as number of cells at each level, modeled after the corresponding private cluster.

Figure 3 shows an example, where there are 4 levels of cell structures: at the GPU (level-1), PCIe switch (level-2), CPU socket (level-3), and node levels (level-4), respectively. The cluster has one rack that consists of four 8-GPU nodes, shared by three tenants, *A*, *B*, and *C*. The cell assignment for each tenant's VC is summarized in the table in Figure 3. Tenants *A* and *B*'s VCs both reserve one level-3 cell (4 GPUs under the same CPU socket), one level-2 cell (2 GPUs under the same

[1] The anomaly is dominated by queuing delay in the job completion time when the affinity requirement is hard. Details discussed in §5.1.

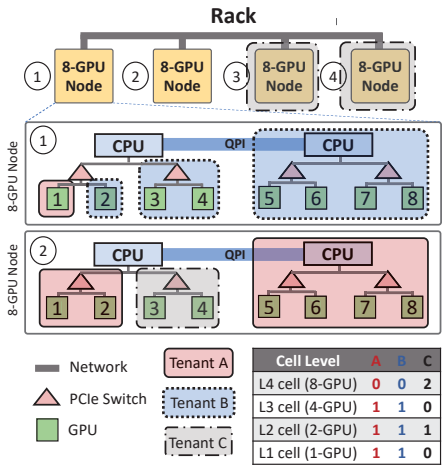


Figure 3: Multi-level cell assignment for a rack: an example.

PCIe switch), and one level-1 cell (single GPU). Tenant *C* is a larger tenant, which reserves two level-4 cells (node level) and one level-2 cell. Given the VC views defined in Figure 3, HiveD can adopt a third-party scheduler [41, 52, 60, 86] to work on the views. From the third-party scheduler's point of view, the VC view is no different from a private cluster consisting of nodes with different sizes (i.e., different level of cells). For example, the scheduler can treat tenant *C* as a private cluster with two 8-GPU nodes and one 2-GPU node, despite the fact that the 2-GPU node is actually a level-2 cell. Note that a third-party scheduler can use any GPUs in the assigned cells. For example, it can schedule two 2-GPU jobs to a 4-GPU (level-3) cell: a cell is the granularity of resource reservation in VCs and the physical cluster, but not necesarily the job scheduling granularity of a third-party scheduler.

In the cell hierarchy, a level-k cell c consists of a set S of level-$(k-1)$ cells. The cells in S are called *buddy cells*; buddy cells can be merged into a cell at the next higher level. We assume cell demonstrates *hierarchical uniform composability*: (i) all level-k cells are equivalent in terms of satisfying a tenant request for a level-k cell, and (ii) all level-k cells can be split into the same number of level-$(k-1)$ cells.

Heterogeneity. A heterogeneous cluster can be divided into multiple homogeneous ones satisfying hierarchical uniform composability. This is logical in practice because a production cluster typically consists of sufficiently large homogeneous sub-clusters (each often a result of adding a new GPU model and/or interconnect) [52]. Users typically use homogeneous GPUs for a job for better performance and specify the desired GPU/topology type (e.g., V100 vs. K80).

Initial cell assignment. A cluster provider must figure out the number of cells at each level to be assigned to each tenant's VC. A VC assignment is *feasible* in a physical cluster if it can accommodate all cells assigned to all VCs; that is, there exists a one-to-one mapping from the logical cells in each VC to the physical cells in the physical cluster. The initial cell assignment for VCs depends on factors like budget, business priority, and workload, thus it is handled outside of HiveD (§6 for further discussion). A cluster might spare more physical resources than the assigned cells to handle hardware failures.

Note that dashed lines in Figure 3 illustrate only one possible cell binding. HiveD advocates dynamic cell binding for flexibility, which reduces job preemption and fragmentation of GPU affinity. §5.3 confirms its benefits over static binding.

Algorithm 1 Buddy Cell Allocation Algorithm

```
1:  // Initial state of free_cells: only top level has cells
2:  procedure AllocateCell(cell_level)
3:    if free_cells[cell_level].size() == 0 then
4:      c = AllocateCell(cell_level+1)
5:      cells = Split(c)                    ▷ Split cells are buddies
6:      free_cells[cell_level].extend(cells)
7:    Return free_cells[cell_level].pop()
8:
9:  procedure ReleaseCell(cell)
10:   if cell.buddies ⊆ free_cells[cell.level] then
11:     higher_cell = Merge(cell, cell.buddies)
12:     free_cells[cell.level].remove(cell.buddies)
13:     ReleaseCell(higher_cell)
14:   else
15:     free_cells[cell.level].add(cell)
```

3.3 Buddy Cell Allocation Algorithm

HiveD manages the dynamic binding between the logical cells in VCs and the physical cells in the physical cluster, and handles requests to allocate and release cells. This is done by the *buddy cell allocation* algorithm. The algorithm maintains for each VC the information of (i) the corresponding physical cell for each allocated logical cell (i.e., the binding); (ii) a global free list at each cell level k to track all unallocated physical cells at that level. The algorithm always keeps available cells at the highest possible level: for example, if all the buddy cells at level-$(k-1)$ are available for a cell at level-k, only the cell at level-k is recorded. And the algorithm aims to keep as many higher-level cells available as possible. Algorithm 1 shows the pseudo-code of the algorithm.

To allocate a level-k cell in a VC, the algorithm starts at level-k and goes up the levels if needed: it first checks whether a free level-k cell is available and allocates one if available. If not, the algorithm will move up level-by-level, until a free level-l cell is available, where $l > k$. The algorithm will then split a free level-l cell recursively into multiple lower-level cells, until a level-k cell is available. Each splitting produces a set of buddy cells at the next lower level, which will be added to the free list at that lower level. One of those new low-level cells is again split until free level-k cells are produced.

The cell release process also works in a bottom-up manner. When a level-k cell c is released, the algorithm adds c into

the free list of level-k cells and checks the status of c's buddy cells. If all of c's buddy cells are free, the algorithm will merge c and its buddy cells into a level-$(k+1)$ cell. The merge process continues recursively while going up the levels, until no cells can be merged. In this way, the buddy cell allocation algorithm reduces GPU fragmentation and creates opportunities to schedule jobs that require higher-level cells.

Before processing an allocation request, the algorithm ensures the request is *legal* in that it is within the assigned quota for the VC at this cell level. HiveD stores the cell assignment in a table r, where a tenant t's preassigned number for level-k cells is stored in $r_{t,k}$. The buddy cell allocation algorithm guarantees to satisfy all legal cell requests under a feasible initial VC assignment, which is formally stated in Theorem 1.

Theorem 1. *Buddy cell allocation algorithm satisfies any legal cell allocation, under the condition of hierarchical uniform composability, if the original VC assignment is feasible.*

Proof. Denote as $r_{t,k}$ the number of level-k cells reserved by tenant t, i.e., cell assignment for t. Denote as r_k the number of reserved level-k cells for all tenants, i.e. $r_k = \sum_t r_{t,k}$. Denote as $a_{t,k}$ the number of level-k cells that have already been allocated to t by the buddy cell allocation algorithm. Cell allocations that maintain $a_{t,k} \leq r_{t,k}$ are legal. Denote as a_k the number of allocated level-k cells for all tenants (i.e., $a_k = \sum_t a_{t,k}$), and f_k the number of free level-k cells in the physical cluster, and h_k the number of level-$(k-1)$ buddy cells that a level-k cell can be split into (hierarchical uniform composability). Define F_k as the number of level-k cells that can be obtained by splitting the higher level cells while still satisfying the safety check for the cell assignment. F_k can be calculated by Eqn. (1).

$$F_k = \begin{cases} (f_{k+1} + F_{k+1} - (r_{k+1} - a_{k+1}))h_{k+1} & k < \hat{k}; \\ 0 & k = \hat{k}, \end{cases} \tag{1}$$

where $\hat{k}$ is the highest level.

To prove the theorem, we prove the following invariant:

$$r_k - a_k \leq f_k + F_k \qquad \forall k = 1, 2, ..., \hat{k}. \tag{2}$$

The L.H.S. is the number of level-k cells all tenants have yet to allocate, and the R.H.S. is the number of available level-k cells the cluster can provide.

We prove by induction on discrete time slots. Denote as w the sequence number of time slots. A change of the cluster state will increase w by 1. When $w = 0$, $a_k = 0$, the invariant (2) holds as long as the original VC assignment is feasible. Assuming the invariant holds at time $w = i$, we shall prove the invariant still holds at time $w = i+1$ after a tenant allocates a legal level-k cell.

Because the allocation is legal, $a_k < r_k$ should hold at time $i+1$. In order to satisfy the invariant (2), either $f_k > 0$ or $f_k = 0$.

When $f_k > 0$, according to Algorithm 1, $a_k = a_k + 1$ and $f_k = f_k - 1$ after an allocation of level-k cell at time $i+1$. The gap of both sides in the invariant remains constant, thus it still holds.

When $f_k = 0$, i.e., no free cell at level-k, the algorithm will split a level-k' cell by finding the smallest k' where $k' > k$ and $f_{k'} > 0$. In this case, the invariant remains true as in the $f_k > 0$ case, while the gap of the invariant at level-k' will decrease by 1. If the invariant at the level-k' breaks after cell splitting, it would mean $r_{k'} - a_{k'} = f_{k'} + F_{k'}$ at time $w = i$. By definition, F_k should be 0 at time $w = i$. But since $a_k < r_k$ (because the allocation request is legal), thus the invariant (2) cannot hold true at level k. This leads to a contradiction. Therefore, the invariant must hold at level k' after splitting a level-k' cell. Following the same step, we can prove the invariant holds at level k'' when the algorithm recursively splitting a level-k'' cell, where $k'' \in [k+1, k'-1]$. Hence the invariant holds on all levels when $f_k = 0$.

Merging the buddy cells can only either increase or keep the gap of the invariant and thus it still holds. Q.E.D. □

The buddy cell allocation algorithm has the time complexity of $O(\hat{k})$, where $\hat{k}$ is the number of levels, and can therefore scale to a large GPU cluster efficiently: $\hat{k}$ is usually 5, from the level of racks to the level of GPUs.

Hierarchical uniform composability ensures the algorithm's correctness and efficiency: it does not have to check explicitly after each split whether or not the subsequent legal allocation requests are satisfiable. Instead, it just needs to check whether every allocation request is legal. For the case where cells are heterogeneous (e.g., due to different GPU models or different inter-GPU connectivities), HiveD partitions the cluster into several pools within which cells at the same level are homogeneous, and applies Algorithm 1 in each pool.

The algorithm resembles buddy memory allocation [56], hence the name. Beyond reducing fragmentation efficiently [35], our key contribution here is making the non-obvious observation: GPU affinity can be modeled as cells, thus making buddy allocation applicable. Moreover, we prove that buddy cell allocation satisfies sharing safety, while traditional buddy allocation does not have such safety concern and hence does not provide this guarantee. Our algorithm also reveals the different characteristics of GPU hierarchy vs. memory regions; for example, the hierarchical uniform composability condition captures GPU hierarchy and is a generalization of the artificially-created power-of-2 rule in buddy memory allocation. Our algorithm also supports priority (elaborated next).

Allocating low-priority cells. The buddy cell allocation algorithm can be naturally extended to support low-priority jobs (a.k.a. opportunistic jobs), whose allocated cells can be preempted by high-priority jobs. Supporting such low-priority jobs helps improve overall GPU utilization, without compromising the sharing safety guarantees provided to the

VCs. HiveD maintains two cell views, one for allocating high-priority (guaranteed) cells, and the other for allocating the low-priority cells. Both views manage the same set of cells in the physical cluster using the same cell allocation algorithm (i.e., Algorithm 1). Similar to YARN [83] and Omega [73], HiveD enforces strict priority where high-priority bindings can preempt low-priority cells. Note that preempting a low-priority job could lead to loss of training progress if its checkpoint is stale. When allocating low-priority cells, HiveD chooses the cells farthest away from those occupied by high-priority jobs (e.g., a non-buddy cell of a high-priority cell) in order to minimize the chance of being preempted. Likewise, when allocating high-priority cells, HiveD chooses the free cells with the fewest GPUs used by low-priority jobs to reduce the chances of unnecessary preemptions. With a similar approach, we can extend HiveD to support multiple levels of priority.

HiveD adopts weighted max-min fairness [37,49] to decide the numbers of low-priority cells allocated to tenants. One could incorporate other state-of-the-art fairness metrics [60] to decide the fair share among tenants.

4 Implementation

HiveD has been integrated in OpenPAI, an open-source deep learning training platform [20] based on Kubernetes [28]. It has been deployed to multiple GPU clusters, managing various types of GPUs from NVIDIA Volta [19] to AMD MI50 [14]. This includes a cloud cluster with 800 heterogeneous GPUs (200 Azure GPU VMs) where HiveD has been running reliably for 12+ months (as of Nov. 2020). HiveD has served research and production workloads at scale, ranging from long-lasting training of large NLP models (e.g., BERT large [34]) to AutoML experiments that consist of hundreds of short-lived 1-GPU jobs. Next we share our experience in implementing and operating HiveD.

HiveD is implemented in 7,700+ lines of Go codes. In addition, it has a few more thousands of lines of JavaScript, Shell scripts, and YAML specifications to integrate with the training platform. It is implemented as a *scheduler extender* [9], a standalone process that works in tandem with the Kubernetes default scheduler (kube-scheduler [7]). This way, HiveD is able to reuse kube-scheduler's basic scheduling logic.

Cell specification. HiveD relies on a cell specification to understand the cell hierarchies in a cluster and the cell assignments for VCs. Figure 4 presents an example specification for a heterogeneous GPU cluster with two racks of NVIDIA V100 GPUs and one rack of NVIDIA P100 GPUs. `cellHierarchy` describes the two types of multi-level cell structures. `physicalCluster` specifies the cell layout in a physical cluster: two V100 racks and one P100 rack, and their IP addresses. With `physicalCluster` and `cellHierarchy`, `vcAssignment` specifies the cell assignment for a VC: the only P100 rack and 4 V100 nodes are assigned to the VC `vc1`.

```
cellHierarchy:
- name: V100-RACK # cell hierarchy for V100 rack
  hierarchy:
  - cellType: V100-GPU  # level-1 cell
  - cellType: V100-PCIe-SWITCH
    splitFactor: 2 # split to 2 level-1 cells
  - cellType: V100-CPU-SOCKET
    splitFactor: 2
  - cellType: V100-NODE
    splitFactor: 2
  - cellType: V100-RACK # level-5 (top-level)cell
    splitFactor: 8
- name: P100-RACK # cell hierarchy for P100 rack
  hierarchy:
  - cellType: P100-RACK # omit lower-level cells
    splitFactor: 8

physicalCluster:
- topLevelCellType: V100-RACK
  topLevelCellAddresses:
  - 10.0.1.0~7
  - 10.0.2.0~8
- topLevelCellType: P100-RACK
  topLevelCellAddresses:
  - 10.0.3.0~7

vcAssignment:
- vc: vc1 #omit other VCs
  cells:
  - subCluster: P100-RACK
    - cellType: P100-RACK
      cellNumber: 1
  - subCluster: V100-RACK
    - cellType: V100-NODE
      cellNumber: 4
```

Figure 4: A simplified cell specification (in `.yaml` format).

A third-party scheduler can leverage the VC view of `vc1` to make scheduling decisions, as if `vc1` is a physical cluster. Our release of HiveD comes with a tool to automatically detect infeasible VC assignments in the specification.

Handling faulty hardware. When multiple free cells are available, the buddy cell allocation algorithm allows HiveD to avoid using faulty hardware. It prefers binding to a healthy cell when possible. When a VC has no other choice, HiveD will proactively bind to a faulty physical cell so that the third-party scheduler in the VC can see the faulty hardware and avoid using GPUs in the cell.

Fault tolerance. The HiveD process itself is also fault-tolerant. It is deployed as a Kubernetes StatefulSet [10] to ensure a single running instance. HiveD maintains several centralized in-memory data structures to keep all the run-time information used for cell allocation (e.g., the free cell list, and the cell allocation list). To reduce overheads, these data structures are not persistent. HiveD partitions and stores the cell binding decision for each pod in its "pod annotation", which is kept reliably by Kubernetes. If a job has multiple pods, the annotation in each pod stores the cell binding decisions for all the pods of the job. When recovering from a crash, HiveD reconstructs all the in-memory data structures like the cell allocation list and the free cell list from the pod annotation in all the running pods. Moreover, with the cell binding decisions stored in pod annotation, HiveD could detect whether or not there are unscheduled pods and resume the scheduling for the unscheduled ones. In case none of the pods of a job gets scheduled when HiveD crashes, the job manager, another single instance StatefulSet, will receive a timeout and resubmit the job. The fault tolerance of the third-party scheduler is handled by the scheduler itself.

Reconfiguration. We observe that a cluster operator may

occasionally change the cell specification on-the-fly to reconfigure a cluster: adding, removing, or upgrading hardware; adjusting cell assignment for a VC. HiveD treats reconfiguration similar to crash recovery. The difference is during a reconfiguration HiveD will check if there is any inconsistency between the old cell bindings in the pod annotations and the new cell specification. For example, the total bound cells from a VC may exceed the new cell assignment. In this case, HiveD will downgrade the jobs with the inconsistent pods to low-priority jobs and preempt them when necessary.

The failure handling and reconfiguration capabilities of HiveD have been tested and verfied on all the deployed OpenPAI clusters. There are occasional hardware issues that require human intervention, e.g., power failures, GPU hardware failures. HiveD handles the decommission and recommission of hardware smoothly. To fully validate its failure handling capability, we run HiveD on an 800-GPU cluster on 200 Azure low-priority VMs [78]. The 200 Azure VMs consist of 125 NC24 [15] (NVIDIA Tesla K80) and 75 NV24 [16] (NVIDIA Tesla M60) series VMs, which could get preempted anytime. HiveD treats a preempted VM as a faulty cell. When a preempted VM resumes, HiveD will re-include it in the cluster just as a faulty cell turning normal. We observe up to 75% of the preemption rate (150 out of 200 VMs) in the cluster. And HiveD handles the preemptions well. When a VM gets preempted, the deep learning job running atop will migrate to other available GPUs or wait in a queue when GPUs are unavailable. The waiting job will get scheduled within one minute when a desired VM resumes from preemption.

5 Evaluation

We evaluate HiveD using experiments on a 96-GPU cluster on a public cloud and trace-driven simulations on a production workload. Overall, our key findings include:

- HiveD eliminates all the sharing anomalies found in all the tested schedulers. Excessive job queuing delay decreases from 1,000 minutes to zero.
- HiveD can incorporate the state-of-the-art deep learning schedulers and complement them with sharing safety, while maintaining their scheduling goals and preserving sharing benefits with low-priority jobs.
- HiveD guarantees sharing safety under various cluster load. In contrast, high cluster load in quota-based scheme can result in 7× excessive queuing delay.
- HiveD's buddy cell allocation algorithm reduces job preemption by 55% with dynamic binding and fragmentation of GPU affinity by up to 20%.

Experimental setup. We collect a 2-month trace from a production cluster of 279 8-GPU nodes (2,232 GPUs). The trace contains 141,950 deep learning training jobs, each specifying its submission time, training time, number of GPUs with the affinity requirement, and the associated tenant. The cluster is shared by 11 tenants. Table 1 shows each tenant's quota assignment in the real deployment and the distribution of a job's GPU number. Please refer to [52] for more details of the trace and its collection and analysis methodology. We run experiments in a 96-GPU cluster deployed on Azure. The cluster consists of 24 virtual machines (NC24 [15]), each with 4 NVIDIA K80 GPUs.

Tenant	1-GPU	2-GPU	4-GPU	8-GPU	≥16-GPU	Total	Quota
res-a	429	14	260	625	40	1,368	0.37%
res-b	18,319	1,593	931	148	238	21,229	0.73%
res-c	3,285	161	716	185	0	4,347	0.73%
res-d	1,754	0	0	0	0	1,754	1.47%
res-e	2,682	110	3,005	0	0	5,797	1.83%
res-f	8,181	88	618	1,337	559	10,783	28.57%
prod-a	227	54	23	1,132	138	1,574	8.79%
prod-b	16,446	67	605	1,344	22	18,484	10.62%
prod-c	4,692	301	1,905	4,415	1,206	12,519	11.36%
prod-d	781	6	545	650	95	2,077	15.75%
prod-e	58,407	532	2,118	959	2	62,018	19.78%
Total	115,203	2,926	10,726	10,795	2,300	141,950	100%

Table 1: Number of jobs with different GPU demands and quota assignment of tenants.

5.1 Sharing Safety: Cluster Experiments

In this section, we examine sharing safety in traditional quota-based scheme and HiveD on the deployed cluster.

Methodology. We collect a 10-day trace from the original 2-month production trace. To approximate the load of the 2,232-GPU cluster on a 96-GPU one, we scale down the number of jobs by randomly sampling from the 10-day trace proportionally (96 out of 2,232). Due to security reasons, we do not have access to the code and data of the jobs. Therefore, we replace the jobs with 11 popular deep learning models in domains of Natural Language Processing (NLP), Speech, and Computer Vision (CV) from GitHub (summarized in Table 2). We mix these models following a distribution of NLP:Speech:CV = 6:3:1, as reported in [86].

We test three state-of-the-art deep learning schedulers: YARN-CS [52], Gandiva [86], and Tiresias [41]. We obtained the source code of Gandiva and Tiresias [11], and use the same implementation in our experiments. YARN-CS is a modified YARN Capacity Scheduler. It packs jobs as close as possible to spare good GPU affinity, similar to [52]. We further refine the preemption policy of YARN-CS: instead of preempting the latest jobs, it preempts low-priority jobs based on the desired GPU affinity requirement. Otherwise, the baseline of YARN-CS will be much worse. To enforce quota in Tiresias, jobs exceeding the quota will get scheduled in a low-priority queue, which is also sorted by Tiresias. For each scheduler, we compare: (i) each tenant running its jobs in a private cluster with the capacity set to its quota; (ii) tenants sharing the

Type	Model	Dataset
NLP	Bi-Att-Flow [74]	SQuAD [68]
	Language Model [90]	PTB [61]
	GNMT [85]	WMT16 [13]
	Transformer [82]	WMT16
Speech	WaveNet [81]	VCTK [12]
	DeepSpeech [45]	CommonVoice [5]
CV	InceptionV3 [79]	ImageNet [33]
	ResNet-50 [46]	ImageNet
	AlexNet [57]	ImageNet
	VGG16 [77]	ImageNet
	VGG19	ImageNet

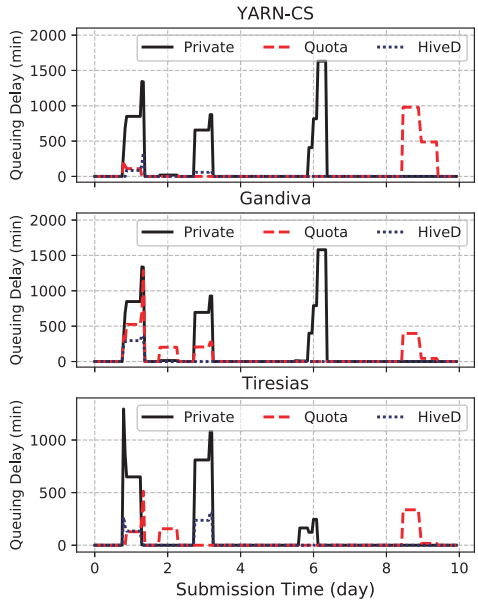
Table 2: Deep learning models used in the experiments [86].

cluster using quota; and (iii) tenants sharing the cluster using the scheduler with HiveD enabled. In a shared cluster, all schedulers will schedule jobs as high-priority ones if the tenant has sufficient resources in its quota or VC, otherwise the job will be scheduled as a low-priority one.

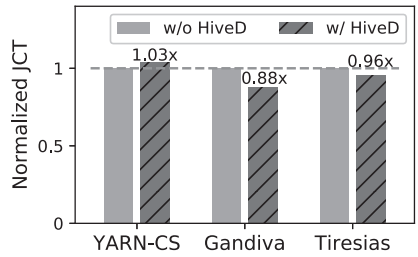
In HiveD's experiments, we use a cell hierarchy with four levels: node (8-GPU), CPU socket (4-GPU), PCIe switch (2-GPU), and GPU. We assign each tenant a set of node-level cells with a total number of GPUs equal to its quota. To model the cell hierarchy after the production cluster, we treat every two contiguous 4-GPU VMs as one logical 8-GPU node (i.e., one 8-GPU node level cell). Similar to [86], to speed up replaying the 10-day trace, we "*fast-forward*" the experiment by instructing running jobs to skip a number of iterations whenever there are no scheduling events, including job arrival, completion, preemption, migration, etc. The time skipped is calculated by measuring job training performance in a stable state. To enable the skipping, HiveD bypasses the kube-scheduler and talks to job pods directly.

The trace shows that the GPU affinity requirements of most jobs are hard, showing that users are not willing to sacrifice training performance. In this case, queuing delay is the major source of sharing anomaly in the overall job completion time (JCT). Note that JCT consists of queuing delay and actual training time, and job training time is highly deterministic as long as GPU affinity is the same [86]. Therefore, we show the queuing delay to illustrate the sharing anomaly when job's GPU affinity requirement is hard. We also evaluate the JCT when job's affinity requirement is soft.

Results. Figure 5(a) shows the queuing delay of jobs from tenant prod-a using the three schedulers. The X-axis denotes the job submission time. The Y-axis denotes the queuing delay averaged in a 12-hour moving window. Figure 5(a) shows that all the three schedulers demonstrate sharing anomaly without HiveD. For YARN-CS, from Day 8 to Day 10, jobs in prod-a suffer 1,000 minutes longer queuing delay in a quota-based cluster than in its private cluster. Although YARN-CS packs jobs as compactly as possible, a large number of 1-GPU jobs from other tenants with varying durations make the available GPUs affinity highly fragmented. As a result, multi-GPU jobs have to wait a long time for the desired affinity. Since the



(a) Average queuing delay of Tenant prod-a



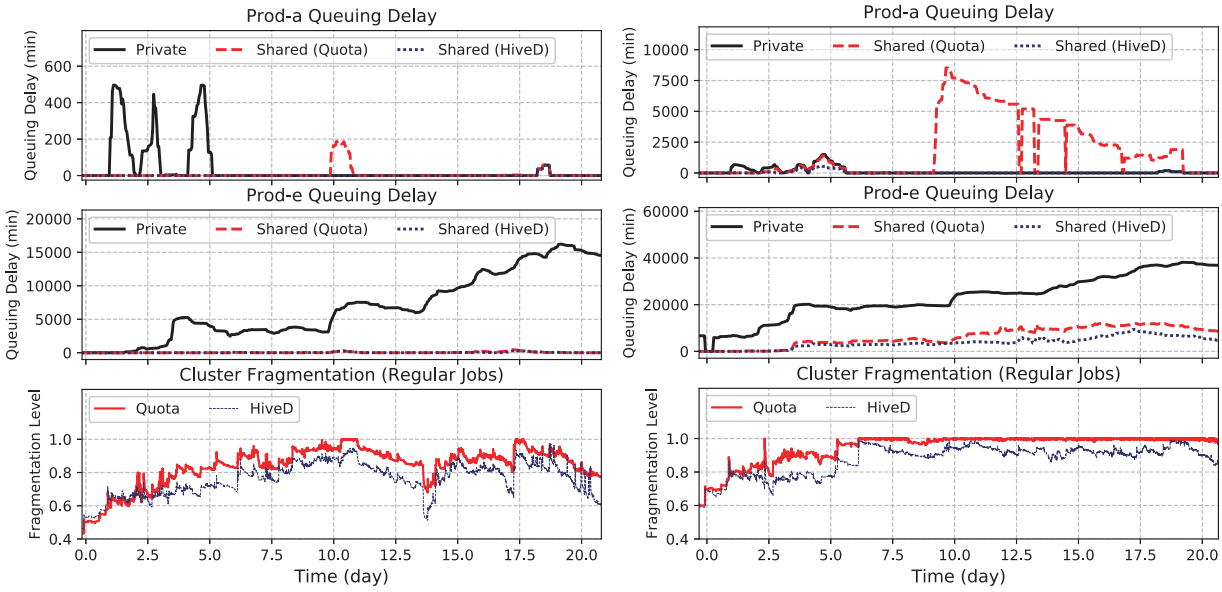
(b) Average job completion time across all tenants

Figure 5: The experiments for the three schedulers in a 96-GPU cluster, with and without HiveD.

majority of jobs in prod-a use multiple GPUs (Table 1), the tenant suffers more from sharing anomaly.

Similarly, in Gandiva, jobs in prod-a suffer up to 400 minutes longer queuing delay in the shared cluster on Day 2 and Day 8. The excessive queuing delay is shorter than that in YARN-CS because Gandiva can mitigate the fragmentation of GPU affinity by job migration. However, unaware of cells in a VC, Gandiva's greedy algorithm may accidentally migrate jobs to improve the job performance in a tenant at the expense of other tenant's GPU affinity, thus violating safety. For example, Gandiva may greedily migrate away an interfering job in a VC while increasing the fragmentation and violating the sharing safety of other VCs. In contrast, HiveD achieves separation of concerns, allowing Gandiva to migrate jobs for its own goal without worrying about sharing safety. We will discuss job migration more in §6.

In Tiresias, Tenant prod-a shows sharing anomaly on Day 2 and Day 8. With quota enforcement, Tiresias suffers over 330 minutes longer queuing delay than that in its private cluster. To reduce job completion time (JCT), Tiresias prefers running



(a) 279 nodes (b) 200 nodes

Figure 6: The average job queuing delay of Tenant prod-a and prod-e vs. the level of fragmentation of GPU affinity.

shorter and smaller jobs first. We do observe shorter queuing delay (and JCT) in Tiresias, compared to the other two schedulers. However, without HiveD, the global advantage of small jobs in a tenant might increase the fragmentation of GPU affinity in other tenants, thus resulting in sharing anomaly.

The experiment suggests that the evaluated schedulers are effective in their design objectives but they do not consider sharing safety, a factor that could severely impact user experience. HiveD complements the three schedulers with sharing safety by reserving the GPU affinity in each tenant's VC. With HiveD, prod-a (and all the other tenants) never experiences an excessive queuing delay in the shared cluster, using each of the three schedulers. Even during Days 8∼10, the multi-GPU jobs are scheduled immediately as the tenant has enough 8-GPU cells in its VC (hence the reserved cells in the physical cluster). HiveD also allows jobs to have a significantly shorter queuing delay in the shared cluster when a tenant runs out of its own capacity in the private cluster (Days 1, 3, and 6), by giving it chances to run low-priority jobs.

With sharing safety, HiveD can still preserve the scheduling efficiency. Figure 5(b) shows that HiveD exhibits similar job completion time compared to those without HiveD: at most 3% worse (for YARN-CS) and 12% better (for Gandiva).

We also evaluate the job completion time (JCT) when job's GPU affinity requirement is soft. Without HiveD, some jobs experience worse training speed due to a relaxed affinity requirement and thus result in higher JCT in a shared cluster than in a private cluster (i.e., sharing anomaly). Again, HiveD eliminates all sharing anomalies in this case. Overall we observe a trend similar to the result when the affinity requirement is hard, hence the details are omitted in this paper.

5.2 Sharing Safety: Full Trace Simulation

We further use simulations to reveal the factors that influence sharing safety. The simulations use YARN-CS as the scheduler in the rest of this section. To validate simulation accuracy, the simulator replays the experiments in §5.1 and we compare the obtained job queuing delay to that in §5.1. The largest difference across all the experiments is within 7%. In the simulations we also observe similar sharing anomalies shown in the real experiments, so we believe the variations do not affect our main conclusion.

Queuing delay in a cluster with the original size. The top two figures in Figure 6(a) show the queuing delay for jobs from two representative tenants, prod-a and prod-e, submitted in 20 days. The jobs run in a cluster of the same size as the original production cluster (279 8-GPU nodes). The result is averaged in a 12-hour sliding window over job submission time. In the bottom figure of Figure 6(a) we also show the level of fragmentation of GPU affinity to observe its correlation with queuing delay. At any time, the level of fragmentation is defined as the proportion of 8-GPU nodes that cannot provide 8-GPU affinity for a high-priority job.

Among the three solutions, HiveD achieves the shortest queuing delay in both tenants. Tenant prod-a suffers a longer queuing delay in its private cluster in several time slots (e.g., the first 5 days) when the resource demands exceed its capacity. Both the quota-based scheme and HiveD reduce the queuing delay significantly by running low-priority jobs. However, from Day 11 to Day 12, prod-a experiences a longer queuing delay (200 minutes) in the quota-based cluster than that in the private cluster. In this period, the fact that no queuing delay

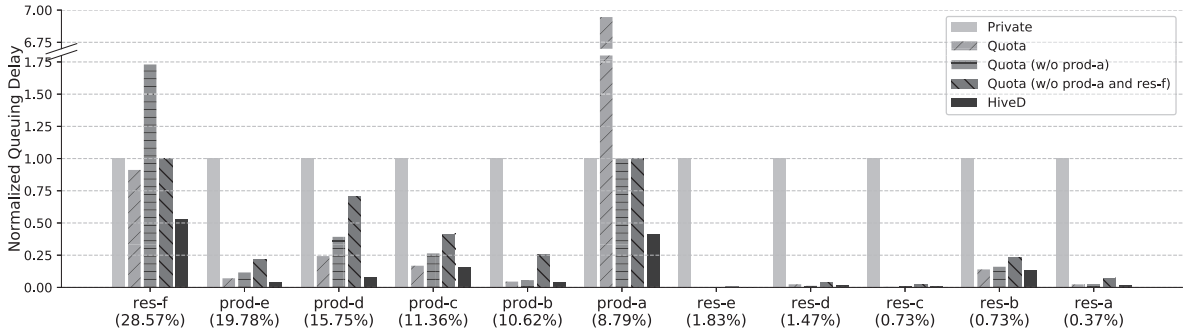


Figure 7: Average queuing delay of each tenant, normalized to that in its private cluster (200 nodes).

observed in its private cluster suggests prod-a has enough GPU quota. But the fragmentation level in the cluster reaches 100%, suggesting the quota-based scheme cannot find even one node to run an 8-GPU job for prod-a. In comparison, prod-a in HiveD has zero queuing delay since it has enough 8-GPU cells available. Overall, the fragmentation level in HiveD is lower than that in the quota-based scheme, because HiveD reserves cells for each tenant, preventing the fragmentation of reserved GPU affinity.

Queuing delay in a higher-load cluster. When a cluster is under-utilized, sharing anomaly is less likely to happen due to sufficient GPU affinity. To further understand the impact of cluster load on sharing safety, we keep the workload unchanged but reduce the cluster size to 200 8-GPU nodes (1,600 GPUs) and rerun the simulation. In this setup, around 90% of the GPUs are used by high-priority jobs. The results are shown in Figure 6(b). In the quota-based scheme, prod-a experiences more severe sharing anomaly when the cluster load is higher. The anomaly lasts from Day 9 to Day 19: the queuing delay can be 8,000 minutes longer than that in the private cluster. The higher cluster load incurs a higher level of GPU affinity fragmentation: the fragmentation level stays at 100% for most of the time, which delays the multi-GPU jobs.

For tenant prod-e, the queuing delays for both Quota and HiveD are always shorter in a shared cluster than in the private cluster. This is because its workload is dominated by a large number of 1-GPU jobs (refer to Table 1), which are immune to the fragmentation of GPU affinity. HiveD can further reduce prod-e's queuing delay by guaranteeing its multi-GPU affinities for its multi-GPU jobs.

We also compare the average queuing delay in the three schemes for each tenant and show the result in Figure 7. The bars marked "Private" and "Quota" show that prod-a's queuing delay in Quota is nearly 7× that in its private cluster. In contrast, the bars marked "HiveD" show that every single tenant has a shorter queuing delay in HiveD than in the private cluster. Compared to Quota, HiveD reduces the queuing delay in 9 out of the 11 tenants (accounting for over 98% quota) due to lower fragmentation level. This reduction is up to 94% (for tenant prod-a), and on average 9% for all the 11 tenants.

In all the previous experiments, the cluster utilization in HiveD is similar to or slightly better than that in quota-based scheme. At some time instances, HiveD improve the utilization over quota-based scheme by up to 20% in the 200-node case and 14% in the 279-node case, as a result of reduced queuing delay. In fact, cluster utilization may depend on the "shape" of jobs (i.e., number of GPUs per job). For example, with a sufficient number of 1-GPU jobs, one can easily saturate the whole cluster. Therefore, our evaluation does not focus on cluster utilization.

Sharing anomalies leading to diminishing benefits of sharing. Figure 7 shows prod-a suffers from severe sharing anomaly (7× queuing delay). It is no longer beneficial for prod-a to contribute its GPUs to the shared cluster. We then run the experiment again to evaluate the effect of decommissioning prod-a (removing its GPUs and workload) from the cluster. The result is shown in the bars marked "Quota (w/o prod-a)" in Figure 7. This time, res-f becomes the victim of sharing anomalies, suffering over 1.7× longer queuing delay. As the largest tenant, res-f previously benefits less (9% shorter queuing delay in Quota than its private cluster) from contributing GPUs to the cluster, compared to the smaller tenants. Because prod-a contains mostly multi-GPU jobs, after decommissioning prod-a, the fragmentation of GPU affinity in the whole cluster becomes worse, leading to longer queuing delay of res-f's multi-GPU jobs and hence the sharing anomaly. This experiment shows the importance of ensuring sharing safety for large tenants. They already benefit less from the shared cluster. They will prefer not contributing their resource to the cluster if experiencing sharing anomaly.

We further decommission res-f from the cluster and rerun the experiment. The result is shown in the bars marked "Quota (w/o prod-a and res-f)" in Figure 7. This time, we do not discover further sharing anomaly. However, the decommissioning of the two tenants greatly reduces the sharing benefits of other tenants. prod-a and res-f contribute 37% of the GPUs in the original cluster. The queuing delay of other tenants in the smaller cluster is clearly longer than that in the larger clusters (before removing prod-a and res-f).

In contrast, with HiveD, not a single tenant suffers from

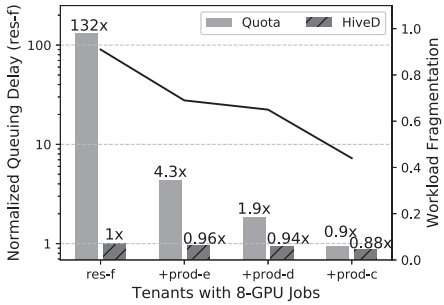


Figure 8: Queuing delay of res-f normalized by private cluster vs. workload fragmentation level (200 nodes).

sharing anomaly in all the settings. And HiveD's queuing delay is consistently shorter than those without HiveD, across all settings. This highlights the necessity of sharing safety.

GPU affinity requirement vs. Sharing safety. We find that the distribution of the GPU affinity requirement for jobs across tenants affects sharing safety. Large number of 1-GPU jobs from other tenants may interfere (or fragment) the GPU affinity of a tenant, leading to sharing anomaly. To show this, we "reshape" the GPU affinity requirement of jobs and observe the queuing delay. In the experiment, we divide the 11 tenants into two groups: jobs in one group are reshaped to the GPU affinity of 1×8 (8-GPU), and those in the other group are changed to 1×1 (1-GPU). The reshaping does not change the total number of GPUs used in each tenant: the number of jobs N is defined by the total number of GPUs divided by the GPU number of a job (8 and 1 in this case). And the job submission time is set by randomly sampling N jobs from the original trace. We further define *workload fragmentation* as the ratio of the total number of jobs to the total number of GPUs. Jobs with higher affinity level have a lower workload fragmentation. The metric will be 1 if all jobs use 1 GPU.

Initially, only res-f is in the 8-GPU group, while the rest tenants go to the 1-GPU group. Figure 8 shows the queuing delay of res-f (normalized by that in its private cluster) and the workload fragmentation, when tenants prod-e, prod-d, and prod-c are moved to the 8-GPU group one by one. When only res-f is in the 8-GPU group, the workload is highly fragmented (0.91). This leads to severe sharing anomaly in Quota-based system: $132\times$ of the queuing delay in the private cluster. When more tenants are moved to the 8-GPU group, the workload fragmentation level goes down. This correspondingly reduces sharing anomaly. res-f experiences shorter queuing delay after the other three tenants are added ($4.3\times$, $1.9\times$, $0.9\times$, respectively, as shown in Figure 8). In contrast, HiveD guarantees sharing safety even under the highest fragmentation and consistently provides shorter queuing delay. Similar trends are observed in other tenants, we hence omit the detailed results here.

Soft affinity requirement. We also study the impact on sharing safety when the GPU affinity requirement of some jobs is soft, i.e., relax the affinity if it cannot be satisfied.

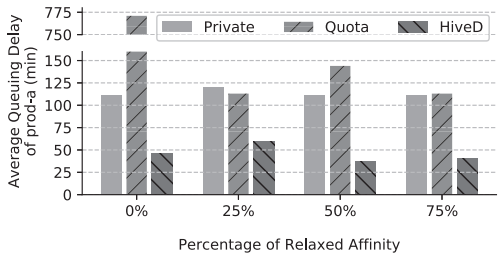


Figure 9: Sharing anomaly still happens when some jobs' GPU affinity requirement is soft.

According to [86], not all training jobs will suffer from performance degradation with relaxed affinity. Hence in the study, we make the most optimistic assumption on the performance degradation: jobs with soft affinity requirement will not sacrifice the training speed. Surprisingly, sharing anomaly could still happen in this case for a quota-based scheme. Figure 9 shows the average queuing delay of tenant prod-a when some multi-GPU jobs in the trace are randomly selected to relax its GPU affinity. We use the 200-node setting in the experiments.

Figure 9 shows that prod-a still has sharing anomaly when 50% of the multi-GPU jobs are allowed to relax their affinity. The average queuing delay in the quota-based scheme is $1.3\times$ of that in its private cluster. Although no obvious anomaly found in the average queuing delay when the job ratio set to 25% and 75%, we still observe sharing anomalies in certain time instances. This is similar to the behaviors in Figures 5(a) and 6. We omit the details due to space limit. On the other hand, HiveD eliminates all the sharing anomalies and always has the shortest queuing delay. Note that Figure 9 shows the best case scenario for relaxed affinity. In reality, jobs with relaxed affinity could perform much worse than the same jobs with the desired affinity [86]. Thus sharing anomaly may happen more likely than it is described in Figure 9.

Although relaxing affinity may reduce the queuing delay for jobs with soft affinity requirement, the behavior may increase the fragmentation of GPU affinity in the cluster. This in turn will increase the queuing delay for jobs with hard affinity requirement. It becomes a complex tradeoff among queuing delay, fragmentation of GPU affinity, training performance, and cluster utilization. HiveD reserves cells to achieve sharing safety and avoids the complex tradeoff altogether.

5.3 Buddy Cell Allocation

In this section, we evaluate the buddy cell allocation algorithm through trace-driven simulations, to understand its effectiveness in reducing preemption and fragmentation of GPU affinity, and its algorithm efficiency.

Reducing preemption with dynamic binding. In the buddy cell allocation algorithm, cells are bound to those in the physical cluster dynamically. This reduces unnecessary preemption of low-priority jobs when there are idle cells. Figure 10 shows the numbers of job preemption when using dynamic

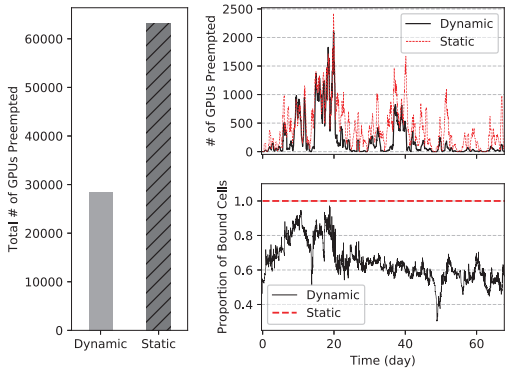


Figure 10: Preemption in dynamic and static bindings.

binding and static binding, respectively. This experiment uses the same setup as the 279-node experiment in §5.2. In total, dynamic binding reduces the number of preempted GPUs by 55%. We also measure the correlation between preemption and the proportion of bound cells in the time dimension (on a 12-hour window). When there are more cells being bound to the physical cluster (e.g., Day 10, Day 20 in dynamic binding), there are also more GPUs being preempted. This is because we have fewer choices of physical cells to bind, hence fewer opportunities to reduce preemption. This observation is also consistent with the fact that static binding, where this proportion is always 100%, incurs many more unnecessary preemptions.

Reducing fragmentation of GPU affinity with multi-level cells. Multi-level cells allow the buddy cell allocation algorithm to pack the cells at the same level across tenants to reduce the fragmentation of GPU affinity. For example, if two tenants both have a level-1 (1-GPU) cell, the algorithm prefers selecting two cells from the same physical node, i.e., buddy cells, to run a 1-GPU job. Instead, if both tenants only reserve level-4 cells (8-GPU, node level), the two tenants have to use a level-4 cell to run its 1-GPU job. Hence the two 1-GPU jobs will be placed on two different nodes, which increases the fragmentation of GPU affinity at node level.

To demonstrate this, instead of only assigning level-4 cells, we assign cells from level-1 to level-4 while keeping the total number of GPUs assigned to each tenant the same as in the above 279-node simulation. Each tenant's assignment matches the distribution of its demands on each level of the cells. Figure 11 shows the fragmentation level of GPU affinity over time when using multi-level and single-level (level-4) cells, respectively. The fragmentation level is always lower with multi-level cells. The gap is more than 10% (up to 20%) for most of the time, which means we can spare roughly 30 more level-4 cells. HiveD therefore recommends that tenants model their job's affinity requirements more precisely, in order for a cluster to perform more efficient cross-VC packing.

Algorithm efficiency. We profile the performance of our

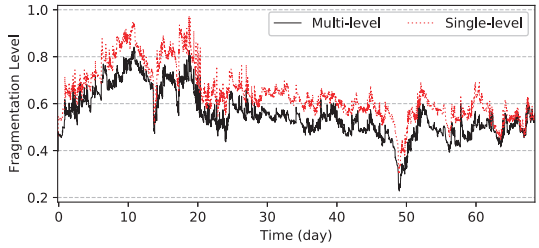


Figure 11: Fragmentation with multi- and single-level cells.

implementation of buddy cell allocation in a setup of a 65,536-GPU cluster with 8 racks, each consisting of 1024 8-GPU nodes. We issued 10,000 cell allocation requests at random levels. The average time to complete a request is 2.18ms. A large part of the cost comes from ordering cells according to low-priority jobs, which accounts for 88% of the time. As the algorithm is clearly not the system bottleneck, we do not perform further optimization (e.g., lock-free operations).

6 Discussion

VC assignment. The VC assignment to a tenant, in terms of both the number of GPUs and their cell structures, has impacts on the effective VC utilization and queuing delay across tenants. The VC assignment is usually a business process, a common practice in large production clusters, e.g., Borg [84]. Factors to consider in VC assignment include overall capacity, tenant demands, composition of tenant workload, workload variation over time, business priority, and budget constraints. Therefore, HiveD leaves the choice of VC assignment to users. In most cases, a tenant can just reserve several node-level cells as a VC and adopt a deep learning scheduler for the VC. If a tenant has more details about workloads, e.g., the GPU number distribution of the jobs, the tenant can reserve different levels of cells to match the job requirement and enjoy less fragmentation and preemption, as discussed in §5.3. VC assignment is a new kind of resource reservation based on cells, and HiveD is a framework to enforce such a reservation.

Job migration. Migrating jobs between GPUs is a powerful mechanism that has been shown effective [86] in improving quality of GPU allocations. De-fragmentation via migration can in theory be used to resolve potential sharing safety violations, but our experience has shown that there are significant challenges in applying migration in production. Fully transparent migration remains challenging in practice, due to implementation issues in different deep learning frameworks (e.g., inconsistent or limited use of certain programming APIs; challenges of multi-language, multi-framework, and multi-version support [21, 30, 62, 65]). Moreover, the choice of which jobs to migrate and where could be rather complex, with different conflicting objectives to balance and a large search space. As shown in §5.1, a greedy migration algorithm [86] can still violate sharing safety. In contrast, HiveD's cell abstraction and buddy cell allocation algorithm enable separation of con-

cerns. HiveD can also leverage migration, especially within each tenant—it will be a search space constrained to within a tenant under the sharing-safety guarantee.

HiveD in the cloud. Major cloud providers are offering GPU VMs in the cloud. Our findings in HiveD are highly relevant even in the cloud setting and can shed light on the types of offering in the cloud. Our buddy cell allocation algorithm can also be used by the cloud providers to manage their reserved [2,6,27] and spot [4,8,78] GPU instances, as our VC cells are essentially reserved instances and our low-priority cells are essentially preemptible spot instances. HiveD's implementation already satisfies requirements of a typical cloud provider, e.g., supports different GPU models, reserves pay-as-you-go instances [70], and handles expansion in capacity. For practical deployment, HiveD can use a hybrid strategy to leverage the cloud as an extension of a multi-tenant GPU cluster when the demand temporarily exceeds the capacity, or can be deployed entirely on a cloud using reserved resources at a lower price, with the options to (i) use spot instances, (ii) buy pay-as-you-go instances when needed, and (iii) purchase and sell reserved capacity in the marketplace [3,23].

Extending HiveD to other affinity-aware resources. Although this paper focuses on reserving affinitized GPUs, HiveD's design applies to other types of affinity-aware resources as well. For example, the cell can be used to define affinitized CPU cores within the same NUMA node, or even multiple types of NUMA-aware resources like affinitized GPUs and CPU cores under the same socket [18].

7 Related Work

Affinity-aware schedulers for deep learning training. Affinity has been well considered something important when scheduling deep learning jobs [22,41,51,52,59,66,72,86] as well as other (big-data) jobs [36,38,89]. HiveD complements these schedulers by applying them in virtualized cluster views, thereby leveraging their efficiency while avoiding sharing anomalies, as identified and shown in our experiments.

Fairness in shared clusters. Identifying the fair share of resources in large clusters has been widely studied. Max-min fairness [55] has been extended in a CPU cluster to address fair allocation of multiple resource types (DRF [37]), job scheduling with locality constraints [38,39,48,89], and correlated and elastic demands (HUG [31]). There are recent proposals to achieve fairness and efficiency for machine learning workloads [29,60,64].

In contrast, HiveD focuses on sharing safety with respect to *given* resource shares (i.e., VC assignment). As we have discussed in §6, determining the resource shares is usually a business process. HiveD assumes a pre-agreed resource partition among multiple tenants, and enforces it with the sharing safety guarantee. This is driven by witnessing that corporate users are annoyed by the uncertain availability of GPU resources that are already assigned to them. In this sense, HiveD is a framework to guarantee a type of resource reservation [91], defined in terms of cells in VCs. HiveD can address fairness by applying the fairness schemes (e.g., Themis [60]) to determine fine-grained fair-share for jobs within a tenant (or across tenants for low-priority jobs), given the coarse-grained VC assignment enforced by HiveD.

Performance isolation. Performance in a shared cluster is sensitive to various sources of interference, including I/O, network, and cache. There are research works on performance isolation that include storage isolation [32,42,43,80], appliance isolation [24,76], network isolation [44,58,67,75,87], and GPU isolation [25,26,50,53,69,88]. In HiveD, we identify a new source of interference: the fragmentation of GPU affinity in a tenant may affect the GPU affinity in other tenants in a shared GPU cluster. To eliminate such interference, HiveD adopts the notion of VC to encapsulate the requirement in multi-level cells and constrains the scheduling behavior within each VC.

Reducing fragmentation. Reducing fragmentation is important to cluster utilization, which has been widely studied in past decades. Tetris [40] is a multi-resource scheduler to pack tasks to avoid resource fragmentation. Feitelson [35] also proposed a buddy-based algorithm to reduce fragmentation for gang-scheduled jobs in supercomputers. There are also works using migration/preemption to reduce fragmentation for gang-scheduled jobs [63,71,86]. HiveD's buddy allocation algorithm with affinity hierarchy can also effectively reduce fragmentation. More importantly, HiveD takes a step further to guarantee sharing safety, i.e., eliminate the external fragmentation across tenants. Ensuring sharing safety requires not only minimizing fragmentation but also explicitly defining cells assigned to each VC, and enforcing this assignment during physical resource allocation.

8 Conclusion

Motivated by observations from production clusters and validated through extensive evaluations, HiveD takes a new approach to meeting the challenge of sharing a multi-tenant GPU cluster for deep learning by (i) defining a simple and practical guarantee, sharing safety, that is easily appreciated by tenants, (ii) proposing an affinity-aware resource abstraction, cell, to model virtual private clusters, (iii) developing an elegant and efficient algorithm, buddy cell allocation, that is proven to guarantee sharing safety and is naturally extended to support low-priority jobs, and (iv) devising a flexible architecture, to incorporate state-of-the-art schedulers for both sharing safety and scheduling efficiency. All these combined, HiveD strikes the right balance between multiple objectives such as sharing safety and cluster utilization.

Acknowledgements

We thank our shepherd Junfeng Yang and the anonymous reviewers for their constructive feedbacks that helped improve the clarity of the paper. We thank Jim Jernigan and Kendall Martin from the Microsoft Grand Central Resources team for providing GPUs for the evaluation of HiveD. Fan Yang thanks the late Pearl, his beloved cat, for her faithful companion during writing this paper. This work was partially supported by the National Natural Science Foundation of China under Grant No. 61972004.

References

[1] Hadoop: Fair scheduler, 2016. https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[2] Announcing general availability of azure reserved vm instances. https://bit.ly/2jEFKHR, Nov. 2017.

[3] Amazon EC2 reserved instance marketplace. https://aws.amazon.com/ec2/purchasing-options/reserved-instances/marketplace/, Apr. 2019.

[4] Aws spot instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html, Apr. 2019.

[5] Common voice dataset. http://voice.mozilla.org/, Apr. 2019.

[6] Google cloud: Committed use discounts. https://cloud.google.com/compute/docs/instances/signing-up-committed-use-discounts, Apr. 2019.

[7] Kubernetes default scheduler. https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/, June 2019.

[8] Preemptible virtual machines. https://cloud.google.com/preemptible-vms/, Apr. 2019.

[9] Scheduler extender. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/scheduling/scheduler_extender.md, Jan. 2019.

[10] Statefulsets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, June 2019.

[11] Tiresias code. https://github.com/SymbioticLab/Tiresias/, Feb. 2019.

[12] VCTK dataset. https://homepages.inf.ed.ac.uk/jyamagis/page3/page58/page58.html, Apr. 2019.

[13] Wmt16 dataset. http://www.statmt.org/wmt16/, Apr. 2019.

[14] Amd Radeon Instinct MI50 accelerator. https://www.amd.com/en/products/professional-graphics/instinct-mi50, Apr. 2020.

[15] Azure vm: Nc-series. https://docs.microsoft.com/en-us/azure/virtual-machines/nc-series, 2020.

[16] Azure vm: Nv-series. https://docs.microsoft.com/en-us/azure/virtual-machines/nc-series, 2020.

[17] HiveD scheduler. https://github.com/microsoft/hivedscheduler, 2020.

[18] Kubernetes topology manager. https://kubernetes.io/blog/2020/04/01/kubernetes-1-18-feature-topoloy-manager-beta, 2020.

[19] Nvidia v100 tensor core gpu. https://www.nvidia.com/en-us/data-center/v100/, Apr. 2020.

[20] OpenPAI. https://github.com/Microsoft/pai, 2020.

[21] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[22] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 17:1–17:12, New York, NY, USA, 2017. ACM.

[23] Pradeep Ambati, David Irwin, and Prashant Shenoy. No reservations: A first look at amazon's reserved instance marketplace. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

[24] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 233–248, Broomfield, CO, 2014.

[25] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: An application-transparent hardware-software cooperative memory manager for gpus. *arXiv preprint arXiv:1804.11265*, 2018.

[26] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. In *ACM SIGPLAN Notices*, volume 53, pages 503–518. ACM, 2018.

[27] Jeff Barr. Announcing amazon EC2 reserved instances. https://aws.amazon.com/blogs/aws/announcing-ec2-reserved-instances/, Mar. 2009.

[28] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[29] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *EUROSYS*, 2020.

[30] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[31] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 407–424, Santa Clara, CA, 2016.

[32] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[35] Dror G Feitelson. Packing schemes for gang scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer, 1996.

[36] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 4:1–4:13, New York, NY, USA, 2018. ACM.

[37] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.

[38] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. pages 365–378, 04 2013.

[39] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, 2016.

[40] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2015.

[41] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019.

[42] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, volume 9, pages 85–98, 2009.

[43] Ajay Gulati, Arif Merchant, and Peter J Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI*, volume 10, pages 1–7, 2010.

[44] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Conference*, page 15. ACM, 2010.

[45] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[47] Walter L Hürsch and Cristina Videira Lopes. Separation of concerns. 1995.

[48] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[49] Jeffrey Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[50] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041*, 2018.

[51] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *SysML*, 2019.

[52] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[53] Angela H Jiang, Daniel L-K Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A Kozuch, Padmanabhan Pillai, David G Andersen, and Gregory R Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*, pages 29–42, 2018.

[54] Mark S Johnstone and Paul R Wilson. The memory fragmentation problem: Solved? *ACM Sigplan Notices*, 34(3):26–36, 1998.

[55] J. Kay and P. Lauder. A fair share scheduler. *Commun. ACM*, 31(1):44–55, January 1988.

[56] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[58] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM computer communication review*, volume 44, pages 467–478. ACM, 2014.

[59] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 3lc: Lightweight and effective traffic compression for distributed machine learning. *arXiv preprint arXiv:1802.07389*, 2018.

[60] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient gpu cluster scheduling for machine learning workloads. *USENIX NSDI*, 2020.

[61] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

[62] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[63] Ioannis A Moschakis and Helen D Karatza. Performance and cost evaluation of gang scheduling in a cloud computing system with job migrations and starvation handling. In *2011 IEEE Symposium on Computers and Communications (ISCC)*, pages 418–423. IEEE, 2011.

[64] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *OSDI*, 2020.

[65] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[66] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 3:1–3:14, New York, NY, USA, 2018. ACM.

[67] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: sharing the network in cloud computing. *ACM SIGCOMM Computer Communication Review*, 42(4):187–198, 2012.

[68] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[69] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[70] Margaret Rouse. Pay-as-you-go cloud computing. https://searchstorage.techtarget.com/definition/pay-as-you-go-cloud-computing-\PAYG-cloud-computing, Mar. 2015.

[71] Kittisak Sajjapongse, Xiang Wang, and Michela Becchi. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 179–190, 2013.

[72] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[73] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. 2013.

[74] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[75] Alan Shieh, Srikanth Kandula, Albert G Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *NSDI*, volume 11, pages 23–23, 2011.

[76] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012.

[77] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[78] Lee Stott. Microsoft azure low-priority virtual machines – take advantage of surplus capacity in azure, Nov. 2017.

[79] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[80] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[81] Aäron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *SSW*, 125, 2016.

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[83] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[84] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[85] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[86] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[87] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. *ACM SIGCOMM Computer Communication Review*, 42(4):199–210, 2012.

[88] Hangchen Yu and Christopher J Rossbach. Full virtualization for gpus reconsidered. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2017.

[89] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[90] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[91] L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp). https://tools.ietf.org/html/rfc2205#section-2#page-19, 1997. IETF RFC2205.

AntMan: Dynamic Scaling on GPU Clusters for Deep Learning

Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou,*
Zhi Li, Yihui Feng, Wei Lin, Yangqing Jia

Alibaba Group

Abstract

Efficiently scheduling deep learning jobs on large-scale GPU clusters is crucial for job performance, system throughput, and hardware utilization. It is getting ever more challenging as deep learning workloads become more complex. This paper presents AntMan, a deep learning infrastructure that co-designs cluster schedulers with deep learning frameworks and has been deployed in production at Alibaba to manage tens of thousands of daily deep learning jobs across thousands of GPUs. AntMan accommodates the fluctuating resource demands of deep learning training jobs. As such, it utilizes the spare GPU resources to co-execute multiple jobs on a shared GPU. AntMan exploits unique characteristics of deep learning training to introduce dynamic scaling mechanisms for memory and computation within the deep learning frameworks. This allows fine-grained coordination between jobs and prevents job interference. Evaluations show that AntMan improves the overall GPU memory utilization by 42% and computation utilization by 34% in our multi-tenant cluster without compromising fairness, presenting a new approach to efficiently utilizing GPUs at scale.

1 Introduction

Over the past years we have witnessed the great success of Deep Learning (DL) with GPUs. DL already powers several widely-used products today, spreading across fields including computer vision, language understanding, speech recognition, recommendation, advertisement, etc. Therefore, it has become a vital workload integrated into the production pipeline at scale. Large companies often build multi-tenant GPU clusters for DL workloads, similar to shared clusters for big-data analytics.

At Alibaba, we have observed low utilization of GPU hardware in shared multi-tenant DL clusters, while queuing many jobs waiting for resources. Such low utilization of DL cluster arises from two main aspects. Firstly, most DL-production training jobs cannot fully utilize all the GPU resources throughout their execution. Training a DL model often requires a mixture of computations, some of which can hardly be parallelized using GPU, such as graph sampling in graph neural network [21, 54], feature extraction in advertisement [15, 23], data augmentation in computer vision [56], etc. Besides, when scaled to distributed training, 90% of the time can be spent on networking [32]. Secondly, the common reservation-based approach for cluster scheduling results in significant GPU idling because DL jobs often cannot consume partial resources. For example, stochastic gradient descent (SGD) is synchronous and requires all resources to be available simultaneously for gang-scheduling [27]. The cluster scheduler thus forces partially available resources to idle in reserve until the final request is satisfied.

Packing jobs on shared GPUs can boost GPU utilization and make the same cluster accomplish more jobs overall. However, this approach is rarely used in production clusters. The reason is that although improving GPU utilization is beneficial, it is also critical to guarantee the performance of important *resource-guarantee* jobs (*i.e.*, jobs with resource quota). Co-executing multiple jobs on the same GPU can result in interference, which leads to significant performance slowdown of the resource guarantee jobs [48]. What's more, the job packing strategy can introduce memory contention on concurrent jobs, which could even cause the failure of the training jobs if the resource demands of a job abruptly increase. Therefore, it is typical in existing production GPU clusters to perform exclusive allocation of resources on jobs [27].

We present AntMan, a DL system that improves GPU cluster utilization while ensuring fairness and performance of resource-guaranteed jobs by doing cooperative resource scaling to minimize job interference. New mechanisms are introduced in DL frameworks to allocate the exact required amount of GPU memory and computation unit dynamically during the job training. Any spare GPU resources, including GPU memory and compute cycles, could be leveraged by oversubscription jobs. AntMan co-designs the cluster scheduler and DL frameworks to adapt to the inherent fluctuating re-

*Co-first author

source characteristics in production jobs, through framework information aware scheduling, transparent memory extension, and fast continuous inter-job coordination. With this architecture, AntMan opens a space for policy design of co-executing DL jobs using GPU resources. In the GPU clusters of Alibaba, AntMan adopts a simple and practical strategy to maximize the cluster throughput. While providing performance guarantee on *resource-guarantee* jobs, AntMan dispatches *opportunistic* jobs to best-effort utilize GPU resources at a low-priority without any resource guarantees.

We have implemented AntMan by modifying two most popular DL frameworks, PyTorch [35] and TensorFlow [8], to expose necessary new primitives for the cluster scheduler to leverage at runtime. Our scheduling policy is implemented in a scheduler prototype on top of Kubernetes for evaluation, and the complete system is fully implemented in Fuxi [52], the internal scheduler of Alibaba, to serve the production DL jobs in the GPU clusters.

We evaluate AntMan on a 64 V100-GPU Kubernetes cluster to show the advantages of the new scheduling primitives and policies with micro-benchmarks and real workloads. The trace evaluation shows that AntMan can preserve the performance of resource-guarantee jobs ideally without preemption. Moreover, it improves the average Job Completion Time (JCT) of all jobs by up to 2.05x compared to current production cluster scheduler, and 1.84x compared to Gandiva [48], a state-of-the-art DL cluster scheduler. We also deploy AntMan in real production clusters and report the evaluations and statistics on a heterogeneous cluster with over 5000 GPUs. The cluster statistics shows that AntMan improves the overall throughput by offering up to 17.1% more GPUs for DL jobs, significantly reduces the average queuing delay by 2.05x, and raises the GPU memory and computation unit utilization by 42% and 34% respectively.

The key contributions of this paper are as follows.

- We investigate the comprehensive characteristics of production DL clusters to understand low utilization from three aspects: hardware, cluster scheduling, and job behavior (Section 2).
- We introduce two new dynamic scaling mechanisms in both memory and computation unit management for DL frameworks to address the challenges of GPU sharing. The new mechanisms leverage DL job characteristics to dynamically adjust the resource usage of DL jobs efficiently during the job execution (Section 3.1).
- Through co-designing the cluster scheduler and DL frameworks to utilize dynamic scaling mechanisms, we introduce a new industrial method to GPU sharing. This maintains the job service-level agreement (SLA) in a multi-tenant cluster while improving the cluster utilization with opportunistic scheduling (Section 3.2 and 3.3).
- By deploying AntMan in Alibaba to serve tens of thousands of daily jobs, we conduct experiments and report the performance improvement in a cluster with more than 5000 GPUs, demonstrating a productive approach in managing multi-tenant DL cluster fairly and efficiently at scale (Section 5).

2 Motivation

In this section, we start by introducing essential DL terminologies as the background. We then highlight our observations by characterising the GPU production cluster to motivate the design of AntMan. We end by discussing opportunities to leverage the DL training characteristics.

2.1 Deep Learning Training

Deep learning training often consists of millions of iterations, and each iteration processes a few samples, called a *mini-batch*. Usually, a training mini-batch can be divided into three phases. Firstly, samples and model weights are calculated to produce a set of scores, known as a *forward* pass. Secondly, a loss error is calculated between the produced scores and the desired ones using an objective function. The loss is then spread backwards through the model to compute gradients, called a *backward* pass. Finally, the gradients are scaled by a learning rate, as defined by an optimizer, to update the model parameters. The computation output of a forward pass usually includes many data outputs, each of which is called a *tensor*. These tensors should be temporarily held in the memory and consumed by the backward pass to calculate gradients. Usually, to monitor the model quality in training, evaluations are periodically triggered.

To train models with massive data, DL generally adopts data parallelism in multiple GPUs where each GPU is responsible for processing a subset of data in parallel while performing gradient synchronizations per mini-batch before the model update.

In large companies, multi-tenant clusters are commonly used to improve hardware utilization, where users can sometimes oversubscribe GPU resource quota, especially when GPU demands burst [33].

2.2 Characterizing Production DL Cluster

We study resource usage in production clusters from three perspectives: hardware, cluster scheduling, and job behavior.

Low utilization of in-use GPUs. Figure 1 illustrates a one-week statistic of GPU memory usage and computation unit utilization. The numbers are collected from one of the production clusters with thousands of heterogeneous GPUs. GPU memory consumption is normalized by the memory capacity of the running GPU due to the heterogeneity in the GPU

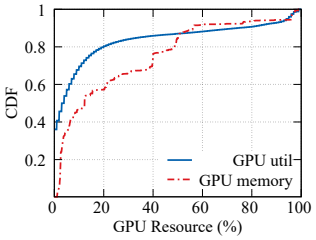


Figure 1: GPU resource statistic on a GPU production cluster.

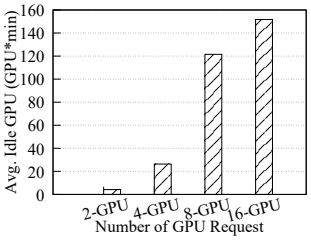


Figure 2: Average GPU idle waiting waste from gang-schedule.

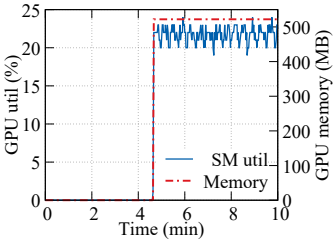


Figure 3: DeepFM on Criteo dataset.

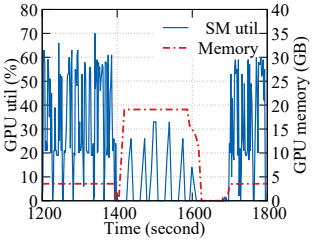


Figure 4: ESPnet on text-speech dataset.

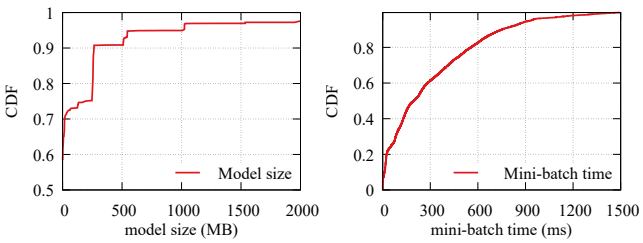
memory capacity. As shown in the figure, only 20% of the GPUs are running applications that consume more than half of the GPU memory. With regards to the usage of computation unit, only 10% of the GPUs achieve higher than 80% GPU utilization. This statistic indicates that both the GPU memory and computation units are not being fully utilized, and are thus wasting the expensive hardware resources.

Idle waiting for gang-schedule. To train deep learning with massive amounts of data, distributed multi-GPU training is essential. Multi-GPU training jobs require gang-scheduling, which means a job will not start training unless all required GPUs are simultaneously available [19, 27]. However, in a cluster, GPU resources can hardly be satisfied simultaneously. (*e.g.*, three GPUs might need to be held and then wait for the last one before launching a 4-GPU job, leaving the three GPUs in idle waiting mode). The more resources a job requires, the more GPU cycles are wasted when in idle waiting mode due to partial resource reservation. To understand the resource waste due to idle waiting, the timestamp of every resource grant for every gang-scheduled job was recorded. The idle waiting time of each GPU (*i.e.*, the gap between the job launching time and the resource granting time) is summed up to calculate the total resources wasted in idle waiting for a job. Figure 2 illustrates the average idle waiting resource waste for different sizes of jobs. The more GPUs a job requires, the higher the cost the cluster must pay for holding idle resources.

The unpredictable arrival of upcoming resources is the reason that reserved resources are left idle. A naïve approach to improving utilization is to launch other jobs on idle waiting resources. However, this can cause the large jobs to become starved and break the scheduling fairness. In addition, once all resources are satisfied, the burst GPU demand of this resource-guarantee job can lead to inter-job resource conflicts with the ones that are currently running in GPUs, which may cause the jobs to fail. Recently, elastic training (*e.g.*, TorchElastic [7]) is proposed to adapt to the incrementally available resources. However, it is rarely used in production because of the non-determinism it introduces to the accuracy [18, 47].

Dynamic resource demand. In addition to the idle wasting from job scheduling, our observation finds that DL jobs usually cannot fully utilize GPU resources during their life cycle. Figure 3 illustrates the first 10 minutes of resource usage when running DeepFM [20] on Criteo dataset. At the beginning, preprocessing on the dataset only requires CPU. However, both GPU Streaming Multiprocessor (SM) utilization and memory usage are boosted at 275 seconds. Such dynamic resource demands also commonly exist in other jobs. Figure 4 illustrates a 10-minute (1200∼1800 seconds) profiling on ESPnet [46], an end-to-end speech model training job. The model training pipeline could contain several phases. During the training phase, ESPnet consumes 3.6 GB GPU memory with a dynamic GPU SM utilization up to 70%. At 1400 seconds, decoding on GPU (around 1400∼1600 seconds) and synthesis (around 1600∼1700 seconds) on the CPU are issued in order to evaluate the model. It is worthy of note that, the decoding phase requires up to 19 GB GPU memory. After the evaluation phase, the model training continues. Such intra-job dynamic resource demand is common in production DL pipelines, making it hard to predict desired resources. We also find some jobs periodically become CPU bound, which is consistent with the observations in neural machine translation tasks [49]. We omit the result due to space limitation.

The dynamic resource demand actually conflicts with the fixed resource allocation and the potentially long running time in the training of deep learning jobs. Jobs requiring sufficient resources according to their peak usage make expensive hardware underutilized. If not granted sufficient resources, the job performance may be limited and thus the job completion time could be delayed. In addition, the memory caching design in existing DL frameworks (*e.g.*, TensorFlow and PyTorch) also conceal the temporal memory usage variations [50], which prevents GPU memory from potential sharing.



(a) Model size distribution. (b) Mini-batch time distribution.

Figure 5: One-week deep learning tasks statistic.

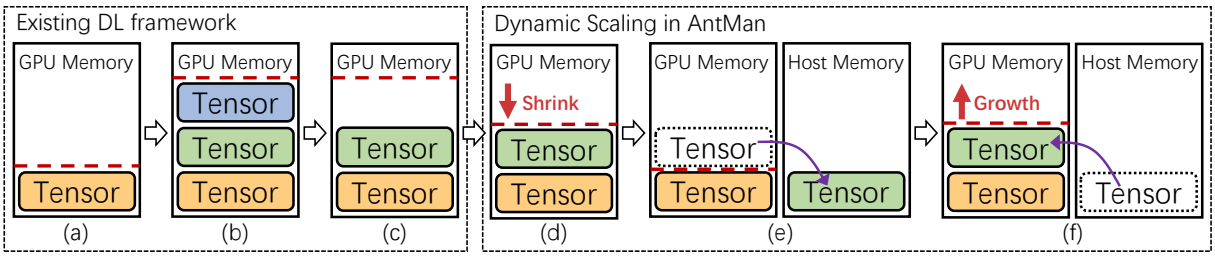


Figure 6: Dynamic scaling universal memory in AntMan

2.3 Opportunities in DL Uniqueness

The preceding characterization of the production DL cluster shows that low utilization is common for both GPU memory and GPU computation unit (*i.e.*, SM). It shows great opportunities to improve the cluster throughput with resource over-subscription. However, the unpredictable inter-job and intra-job demand burst introduces challenges to safe resource sharing. Jobs could run out of memory due to resource contention. Besides, in multi-tenant clusters, it is important to provide performance isolation for jobs holding a resource quota when the jobs are executed in a resource-sharing approach. To cater to these challenges when scheduling deep learning jobs, AntMan leverages the opportunities in the uniqueness of DL training.

We sample 10K tasks in a week of our production cluster to understand DL characteristics. We measure model size and mini-batch size during model training, both shown in Figure 5. Even though DL training could potentially use as much as 32 to 40 GB GPU memory (*e.g.*, V100 and A100), only a small portion is used to store the persistent DL model. 90% of DL models occupy only 500 MB GPU memory.[1] The majority of GPU memory is allocated and freed within the same mini-batch. Moreover, the DL training cycle is also rather small. As much as 80% of tasks consume a mini-batch within 600 *ms*.

We exploit such unique characteristics in several ways to schedule jobs on shared GPUs. Firstly, due to the small model size in common, the majority of GPU memory could be scheduled among the co-executing jobs. Secondly, mini-batch cycles are generally quite small, allowing fine-grained GPU memory and computation scheduling at every mini-batch boundary. This could further allow fast resource coordination between jobs. Thirdly, mini-batches apply mostly similar computations that can be utilized to profile the job performance, therefore their progress rate can be created as a performance metrics to quantify interference.

3 Design

AntMan deeply co-designs cluster schedulers and DL frameworks to address GPU sharing challenges. In this section, we first describe the new mechanism extensions in DL frameworks. We then introduce the collaborative scheduling design to leverage those new primitives. Finally, we present a new productive policy enabled in the cluster scheduler of Alibaba to manage DL jobs.

3.1 Dynamic Scaling in DL Frameworks

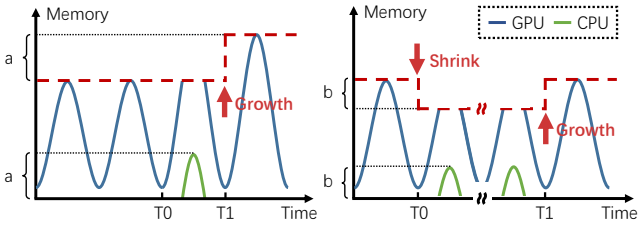
As mentioned in Section 2.2, DL training clusters exhibit low utilization due to unsaturated GPU usage in DL workloads and unique gang-schedule requirements during job scheduling, which contains great potentials that can be exploited to execute more jobs. However, some challenges need to be addressed, such as executing jobs at their minimal requirements while preventing GPU memory usage outbreak failures, adapting to the fluctuating computation unit usage while limiting potential interference. At its core, existing DL frameworks are designed for dedicated GPU executions, which lack key capabilities when collaborating with other jobs. Such conflicts between production DL cluster characteristics and DL framework limitation motivate the design of dynamic scaling mechanisms to enhance DL frameworks. The dynamic scaling mechanisms include the fine-grained dynamic control in two aspects, GPU memory and computation unit. We elaborate them next.

3.1.1 Memory Management

A dynamic memory management mechanism is introduced in AntMan to adapt the allocated memory on the fluctuating memory demands of a DL training job. This is achieved by allocating universal memory to DL application tensors, *i.e.*, switching tensors between GPU and CPU host machine DRAM across mini-batches. Modern operating systems support *paging* in memory management at the granularity of memory pages, where they use disk as memory when they run out of physical memory. AntMan adopts a similar approach, however, this is carried out in an application-specific granularity, tensor, which can be transparently migrated in universal memory addresses at runtime. In this way, DL frameworks can support the dynamic GPU memory upper limit.

Figure 6 illustrates the memory management in existing DL frameworks as well as the differences to AntMan. The total

[1] we omit the largest 2% jobs' model size as the number is business sensitive.



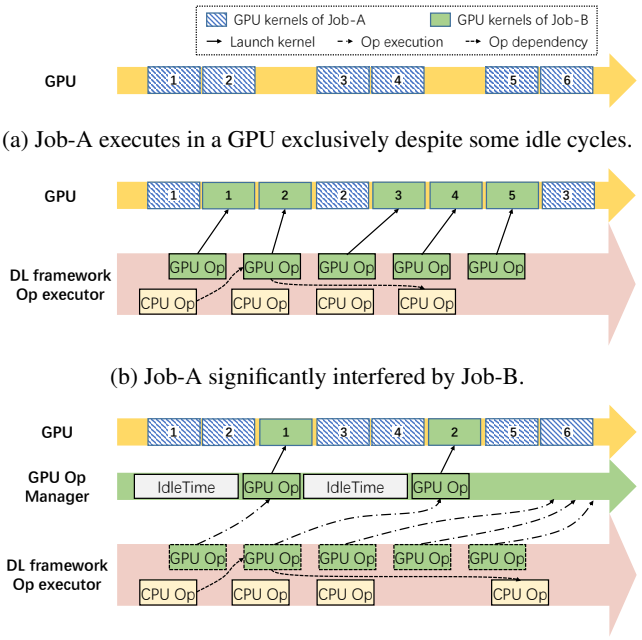
(a) Scaling for memory burst up. (b) Scaling to secure memory.

Figure 7: Leveraging mini-batch behavior to scale memory efficiently.

number of cached GPU memory size (*i.e.*, red dash line) increases with tensors created in DL frameworks (Figure 6a∼b). In order to eliminate the expensive overheads in memory allocations and de-allocations, and also to speed up training among mini-batches, the GPU memory is cached in a global memory allocator inside DL frameworks after tensors are destroyed. Prevalently, some tensors are used only in certain stages of DL training (*e.g.*, data preprocessing, evaluation), which are no longer required. However, this portion of cached GPU memory is not released (Figure 6c). This cached memory design in DL frameworks optimizes individual job performance at the cost of losing sharing potentials.

AntMan turns to the approach of scaling the GPU memory upper limit. It proactively detects in-used memory to shrink the cached memory to introspectively adjust GPU memory usage to an appropriate fit. This is done by monitoring application performance and memory requirements when processing mini-batches (Figure 6d). Furthermore, new primitives are provided to shrink the upper limit of GPU memory at runtime, even below the actual GPU memory demand of a job. AntMan uses its greatest effort to allocate tensors on GPU devices, however, tensors can be allocated outside of GPU with the host memory if GPU memory is still lacking (Figure 6e). With such universal memory support, jobs can continue to process even below their actual GPU memory requirements, where we find workloads slowdown the performance differently (Section 3.3). Tensors can be allocated back to GPU automatically when the GPU memory's upper limit increases (Figure 6f).

Paging in operating systems introduces costly page copy between the memory and disk. In contrast, thanks to the unique pattern of DL, tensor copy between the GPU and CPU host DRAM is explicitly avoided. Identical tensors are created across mini-batches, and therefore, AntMan exploits this pattern to adjust the upper limit of the memory at the boundary of the mini-batches. Figure 7a illustrates how memory scaling addresses the burst demand. At T_0, the memory requirement of a running DL training job increases, due to the limited upper-bound of GPU memory, some tensors cannot be placed in the GPU memory, and are instead created using the host memory. AntMan detects the usage of the host memory, and at T_1, it raises the GPU memory's upper limit for that job according to the usage of the host memory, which allows



(a) Job-A executes in a GPU exclusively despite some idle cycles.

(b) Job-A significantly interfered by Job-B.

(c) GpuOpManager of Job-B controls the interference.

Figure 8: Computation management to run two jobs in a shared GPU without interference.

the tensors to be fully allocated in the GPU device for the next mini-batch. Note that, the performance of this running job might slowdown in a mini-batch as tensors are placed in the host memory. However, such performance overheads are negligible, considering a typical DL training often requires millions of mini-batches. The overhead of memory shrinkage and growth is quantified in Section 5. Furthermore, AntMan provides fine-grained GPU memory scheduling at runtime. A training job might shrink to secure memory resources for other jobs, and grow back after other jobs are finished, as shown in Figure 7b. It illustrates that a DL job scales down at T_0 and scales up at T_1, at the cost of some tensors allocated on the host memory. Therefore, the usage of the remaining GPU memory between T_0 and T_1 for jobs running in the same shared GPU is secured.

3.1.2 Computation Management

Dynamic computation unit management is a mechanism introduced in AntMan to control the GPU utilization of a DL training job. Modern operating systems (*e.g.*, Linux) support cgroups, which limits, accounts for, and isolates the CPU resources that a process requires [1]. AntMan introduces a similar method of dynamically isolating the GPU computation resource access of DL-specific processes at runtime.

When multiple DL jobs are launched on the same GPU, the interference is mainly caused by the potential GPU kernel queuing delay and PCIe bus contention [14], which could

result in consistent performance downgrades across all jobs if packing jobs are running on the same model and configuration [48]. Our observation shows that jobs slowdown in different ways if different jobs are packed together (Section 5.1). This is because jobs have different capabilities at acquiring GPU computation units. Consequently, job performance can barely guarantee or predict in GPU sharing, resulting in difficulties on the deployment of GPU sharing for multi-tenant clusters. Figure 8 illustrates an example of GPU computation unit interference for two jobs that are executed on the same GPU. Figure 8a illustrates how Job-A executes on a GPU in a fine-grained manner. In short, GPU kernels will be placed in order and processed by the GPU computation unit one by one. Note that, in Figure 8, Job-A might not be able to fully saturate the GPU, resulting in idle GPU cycles and low GPU utilization which can potentially be used by other jobs. Therefore, Job-B is scheduled on this GPU (Figure 8b). The GPU operators of Job-B launch kernels (green blocks) executed in the GPU, which can fill it up, and thus delay the execution of other GPU kernels (blue blocks), leading to the poor performance of Job-A. The interference mainly comes from the lack of ability to control the execution frequency of GPU kernels. To address this issue, We introduce a GPU operator manager in DL framework(Figure 8c). Existing DL frameworks issue GPU kernels in the GPU operator once its control dependency is satisfied. In AntMan, the execution of GPU operator is dedicated to a newly-introduced module, called *GpuOpManager*. When a GPU operator is ready to execute, it is added to GpuOpManager instead of being directly launched. The main idea of GpuOpManager is to control the launching frequency by delaying the execution of GPU operators. In this way, AntMan introduces a new primitive to limit the GPU utilization of a DL training job using GpuOpManager. GpuOpManager continuously profiles the GPU operators execution time and simply distributes idle time slots before launching the GPU operators. Note that, GpuOpManager only delays the GPU kernel execution. Therefore, the potential dependencies among operators (including GPU operators and CPU operators) are retained, meaning that CPU operators can continue if possible. As illustrated in Figure 8c, the third CPU operator is not blocked, however, the fourth one is delayed as it depends on the second GPU operator, which has its execution delayed by the GpuOpManager.

3.2 Collaborative Scheduler

In this section, we describe how we co-design the cluster scheduler and DL frameworks to leverage the dynamic scaling mechanisms mentioned above for collaborative scheduling. We focus on the overall architecture of AntMan and how different modules operate. The detailed policy description is in the next section.

As shown in Figure 9, AntMan adopts a hierarchical architecture, where a global scheduler is responsible for job

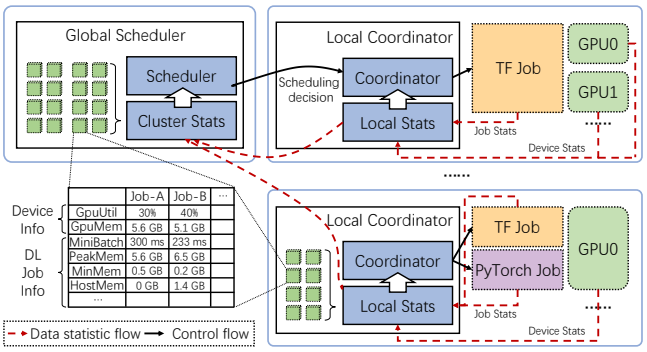


Figure 9: Collaborative scheduling workflow of AntMan.

scheduling. Each working server contains a local coordinator that is responsible for managing the job execution using the primitives of dynamic resource scaling through considering the statistics reported from DL frameworks. AntMan is designed for multi-tenant GPU clusters. In a multi-tenant cluster, each tenant usually owns certain resources, annotated as a resource quota (*i.e.*, number of GPUs), which is the concurrent performance guarantee resources that can be assigned to the jobs of that tenant. The sum of the GPU resource quota of each tenant is less equal to the total capacity of a GPU cluster. In AntMan, jobs are classified into *resource-guarantee* jobs and *opportunistic* jobs by global scheduler with different scheduling policies applied (Section 3.3). Resource-guarantee jobs consume a certain amount of GPU resources quota of their corresponding tenants while opportunistic jobs do not. Therefore, AntMan ensures that the performance of the resource-guarantee jobs should be consistent with that in exclusive executions.

In AntMan, similar to conventional cluster schedulers, the scheduling decision is dispatched from the global scheduler to the local coordinator. In addition, the local coordinator introspectively schedules the GPU resources to DL training jobs using the dynamic scaling mechanisms (Section 3.1). Therefore, the scheduling decisions can be treated as a top-down control flow. In contrast, data statistic flow information is collected by statistic modules of the local coordinator and aggregated on the cluster statistic module in a bottom-up approach to help make scheduling decisions, which is similar to Apollo [10]. Alongside with the hardware information (*e.g.*, GPU utilization, GPU memory usage), AntMan also leverages detailed job information reported by DL frameworks, including mini-batch duration, peak memory usage, minimal memory usage, and host memory consumption, etc. This information can also assist job scheduling decisions made by the global scheduler. For example, peak memory and minimal memory usage are used to indicate the GPU memory size that can be made available quickly. Mini-batch time shows how soon the GPU memory can be available for another DL training job, which can affect the scheduling decisions of the global scheduler when launching jobs.

Algorithm 1 scheduleJob(in job, out nodes)

```
1:  nodes0 ← findNodes(job.gpu, constraints ← job.topo)
2:  nodes1 ← findNodes(job.gpu, constraints ← M)
3:  nodes2 ← minLoadNodes(nodes1, job.gpu)
4:  if job.isResourceGuarantee:
5:      if numGPUs(nodes0) >= job.gpu:
6:          return nodes0
7:      else:
8:          reserve(nodes0)
9:  else:
10:     return nodes2
```

Once a job is launched on a GPU server, a local scheduler takes over the management of its end-to-end execution. Due to the load fluctuation of a DL training job, a local coordinator acts in an introspective mode to perform continual job control to DL frameworks. More specifically, it collects the statistics from the hardware and DL frameworks of all jobs, which is used to control job performance via resource usage adjustments (*e.g.*, shrink GPU memory) through the new primitives we introduced in Section 3.1.

3.3 Scheduling Policy

In this section, we first present the goal of our cluster scheduler. Then we describe the detailed policies applied in global scheduler and local coordinator. Finally, we introduce the job upgrade in our system.

Goal. There is an inherent tension between providing fairness (*e.g.*, to ensure SLAs of DL jobs with guaranteed resources) and achieving high resource utilization (*e.g.*, GPU utilization), because of the constant fluctuation in both the load on a cluster and the resource needs of a job. Prevalent production DL cluster schedulers often trade fairness in certain ways for efficiency. For example, spare resources are allocated to over-provision tenants. However, such GPU resources can hardly get back without preemption. Generally, preemption is rarely used as it fails running jobs while wastes expensive GPU cycles. Besides, [27] also reports the out-of-order behavior which discriminates large jobs (*i.e.*, allocating more GPUs), leading to unfairness by preferring small jobs. In AntMan, multi-tenant fairness is our primary goal, and the second priority is to improve the cluster efficiency therefore to achieve higher throughput. AntMan achieves fairness with the polices that are implemented in both the global scheduler and the local coordinator, powered by the dynamic scaling mechanisms. Furthermore, GPU opportunistic jobs are introduced in AntMan to steal idle cycles in GPUs so as to maximize cluster utilization.

Global scheduler. As a multi-tenant cluster scheduler, the global scheduler maintains multiple queues of tenants where jobs arrive and decides GPU locations allocated for jobs. For resource-guarantee jobs and opportunistic jobs, AntMan applies different scheduling polices as shown in Algorithm 1. *findNodes* is a function that returns the node and GPU candidates which satisfy the job request with an optional parameter to specify constraints. Global scheduler fairly allocates resource-guarantee jobs given sufficient GPU resources. In addition, resource-guarantee jobs are optimized to maximize the job performance using the free GPU resources, *i.e.*, GPUs that are not allocated to other resource-guarantee jobs (line 5-6). For instance, a distributed resource-guarantee job that uses all-reduce communication strategy (*e.g.*, NCCL [5]) can be scheduled on one server to utilize the NVLink [6] for high-performance communication. However, if the resource request of a job can partially be satisfied, the global scheduler reserves the resources for this job, and waits for others to meet the gang-scheduling requirement (line 7-8). Such insufficient resource reservation exists mainly for resource quota (*e.g.*, three GPUs left while there is a request for four) and resource fragmentation (*e.g.*, request four GPUs in the same server, however only four are available spread across servers). The reserved resources will never be occupied by other resource-guarantee jobs, however, they can be utilized by opportunistic jobs.

By default, the global scheduler will estimate the queuing time for jobs without GPU quota granted. Those jobs that suffer long queuing delay will be automatically executed as opportunistic jobs. To schedule opportunistic jobs, global scheduler aims to utilize free resources to the best of its ability. It allocates opportunistic jobs on GPUs by considering the actual GPU utilization, even when some other jobs run on those GPUs. Only GPUs with a utilization of less than M (set as 80% for now) in the past 10 seconds can be selected as candidates. AntMan adopts a heuristic strategy to allocate opportunistic jobs on the freest candidates (*i.e.*, *minLoadNodes*, line 9-10). In this way, there are some jobs allocated on the same GPU, where they are managed by the local coordinator. We will elaborate their coordinated execution next. Note that, although AntMan automatically selects opportunistic jobs by default, it also allows users to manually identify the job type at the point of submission; for example, as a resource-guarantee job explicitly to ensure SLAs. A job can also be specified as an opportunistic job that will never occupy the tenant's resource quota, and vice versa. In practice, users usually submit jobs in opportunistic mode to avoid the potential queuing delay, aiming to perform debugging and hyper-parameter tuning, which are both driven by early feedbacks [48, 51].

Local coordinator. The main responsibility of the local coordinator is to collaborate the execution of jobs on shared GPUs. Next, we first introduce how local coordinator ensures the performance of resource-guarantee jobs at shared execution. Then, we describe the approach to handle resource demand surges of a resource-guarantee job. Finally, we in-

troduce a greedy approach in AntMan to maximize the aggregated job performance when a GPU is only shared by opportunistic jobs. These approaches are achieved by utilizing the information reported from both GPU device and DL frameworks, and by instructing the memory management module (Section 3.1.1) and computation management module (Section 3.1.2) in DL frameworks.

A GPU is allocated to only one resource-guarantee job as it consumes GPU quota. However, in AntMan, it is possible that there are some opportunistic jobs executed on this GPU. As such, the local coordinator must prevent the resource-guarantee job from interfering by other co-located jobs at runtime. When a resource-guarantee job arrives on a GPU that runs with opportunistic jobs, the local coordinator first limits the opportunistic jobs in using GPU, for both GPU memory and GPU SM. By reducing the GPU usage of the opportunistic jobs, the newly launched resource-guarantee job will be capable of persistently initializing the training variables (*i.e.*, model) in the GPU memory. In addition, when launching a DL training job, the GPU device needs to be initialized by the DL framework, which takes more time if the GPU is in a high load. Once the resource-guarantee job is stably executed, the local coordinator will allocate the rest of the GPU memory to the opportunistic jobs. Furthermore, it gradually increases the GPU computation unit usage of opportunistic jobs without interfering with resource-guarantee jobs by monitoring the job performance (*i.e.*, mini-batch time). Similarly, when an opportunistic job arrives on a shared GPU, the local coordinator raises its GPU resource usage in a step-like fashion under the condition that the resource-guarantee job is not affected.

During the job execution, the resource demand of both the GPU memory and GPU computation unit might surge beyond the currently available resources (Section 2.2). To be aware of such dynamic resource demand, the local coordinator monitors the metrics that are reported by DL frameworks (*e.g.*, host memory usage, mini-batch time). Therefore, when a resource-guarantee job increases the GPU memory requirement, the tensors are temporarily stored using host memory, thanks to the universal memory (Section 3.1.1). The local coordinator shrinks the GPU memory usage of other opportunistic jobs and raises the GPU memory limit of the resource-guarantee job to recover its performance. It is similar for GPU computation unit usage coordination. Note that, AntMan relies on the application level metric (*i.e.*, mini-batch time) to indicate the job performance of resource-guarantee jobs. If it observes an unstable performance in the resource-guarantee job, it adopts a pessimistic strategy to limit the usage of GPU resources of other opportunistic jobs.

GPU resources can also be idle waiting without any resource-guarantee jobs (*e.g.*, due to gang-schedule as described in Section 2.2). In this case, if there is only one opportunistic job, the GPU resources can be fully utilized by this job without any constraints. Sometimes, it is possible that a GPU is occupied by multiple opportunistic jobs. Under this

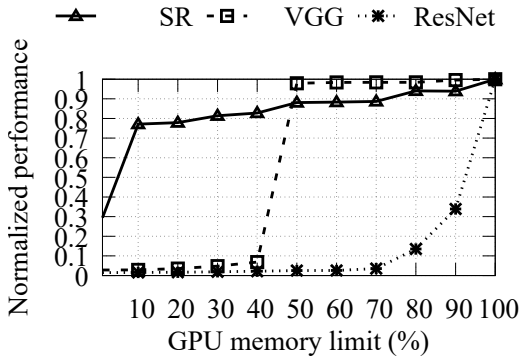


Figure 10: Workloads show diversity in performance sensitivity given insufficient memory.

scenarios, AntMan optimizes the aggregated job performance by maximizing GPU memory efficiency. With the dynamic scaling mechanisms enabled, we find that different workloads show differences in sensitivity regarding the performance slowdown from memory limitations. The peak memory usage of a job is limited using the dynamic memory scaling mechanism, and the host memory is thereby used for the remainder of the excess. As illustrated in Figure 10, Super Resolution (SR) model suffers only around 25% performance slowdown even with a 90% reduction in its device memory. VGG16 [43] model on Cifar10 dataset (VGG) can keep most of its original performance even after reducing its device memory by half. ResNet50 [22] on ImageNet dataset (ResNet) is sensitive to memory shrinkage; a 10% memory reduction introduces more than 60% slowdown. Therefore, when the total GPU memory demand of opportunistic jobs exceeds the GPU's memory capacity, AntMan adopts a simple heuristic approach which allocates GPU memory to the job that improves the normalized aggregated job performance at best. This is carried out via an introspective trial-and-error allocation.

Job upgrade. In AntMan, opportunistic jobs are executed at best-effort level to improve the cluster utilization. However, this is done without an SLA guarantee. The global scheduler upgrades these jobs given sufficient resources to complete them quickly. For distributed synchronous DL training, the partial upgrade does not help because the performance downgrade of a worker can be broadcast to the entire job. Thus, the global scheduler checks if all GPUs are filled up in opportunistic jobs. Once all task instances are ready to upgrade and the resource quota is sufficient, AntMan prefers to upgrade the opportunistic job rather than launch a new one. Global scheduler notifies local coordinator to tag it as a resource-guarantee job and consumes the tenant's GPU quota to accomplish the job upgrade.

4 Implementation

At Alibaba, DL training jobs are executed in Docker containers with our customized versions of DL frameworks. The APIs of the DL frameworks are compatible with the community version however with AntMan's features enhanced. A prototype custom cluster scheduler is implemented on Kubernetes [11] for evaluation. AntMan is fully implemented in our internal cluster scheduler, Fuxi [52], to serve the daily production training jobs on several clusters with thousands of GPUs each.

4.1 Deep Learning Framework

Dynamic scaling mechanisms are implemented in two popular deep learning frameworks, TensorFlow [8] and PyTorch [35], on versions v1.12 and v1.3.1 respectively. The implementation in TensorFlow takes 4000 lines of code (mostly in C++). The implementation in PyTorch takes about 2000 lines of code (500 lines in Python and 1500 lines in C++).

The modification of DL frameworks is mostly in three components: memory allocator, executor, and interfaces. As it adopts a similar implementation in both frameworks, we mainly use TensorFlow terminology to describe the details. To enable dynamic universal memory, `BFCAllocator` (`CUDACachingAllocator` in PyTorch) is modified to introduce an adjustable upper limit for memory. The memory allocator keeps track of the total bytes of memory allocation and triggers out-of-memory when total bytes exceed the upper limit. In addition, a new interface is introduced to the memory allocator to allow emptying of cached memory at any time. A new universal memory allocator, `UniversalAllocator`, is also added to wrap the GPU memory allocator and host memory allocator (*i.e.*, using `cudaHostMalloc` for memory allocation). When a memory allocation is triggered by the request of a tensor, UniversalAllocator tries to allocate the memory using the GPU memory allocator and treats the CPU memory allocator as a backup if there is insufficient GPU memory left over. Note that, the UniversalAllocator maintains a set data structure that records the pointers of memory regions allocated by GPU, which is used to classify the memory pointers for de-allocation.

To enable dynamic computation unit scaling, a `GpuOpManager` with an operator processing queue, which runs in a standalone thread, is introduced in DL frameworks. The operator executor of TensorFlow is modified accordingly to insert GPU operators to GpuOpManager queue in order so as to dedicate the execution of GPU operators to it. GpuOpManager may delay the actual execution of the GPU operators based on a limited percentage of the computation capacity.

The statistics of memory usage patterns and the execution information are aggregated for the local coordinator. The DL frameworks and local coordinator communicate through the file system. They both have a monitor thread to check the file for receiving either job statistics or control signals. To minimize the overhead of memory management, the dynamic scaling of memory is triggered at the mini-batch boundaries (end of `session.run()`).

4.2 Cluster Scheduler

A custom scheduler is implemented on Kubernetes [11] as a prototype to evaluate AntMan. The implementation requires around 2000 lines of code in Python. Overall, Kubernetes is responsible for cluster management and for executing jobs in Docker containers. Our global scheduler uses Python APIs to monitor the events in Kubernetes's API server for scheduling. Local coordinators are deployed as a DaemonSet in Kubernetes. Each coordinator monitors certain paths of the file system to collect the reported information for each job. The aggregated job and device information are stored in ETCD, a built-in distributed key-value store in Kubernetes. Therefore, global scheduler directly reads states in ETCD when making scheduling decisions.

AntMan has been fully implemented in Alibaba's internal cluster scheduler, Fuxi [52]. The implementation of global scheduler takes about 10000 LOC, including failover support and testing. The local coordinator implementation takes about 2000 LOC. Both of them are written in C++. The DL infrastructure is coupled with the big-data infrastructure, as DL jobs are part of the data pipeline. Fuxi adopts an architecture that optimizes for high performance scheduling, and it currently does not have ETCD. Global scheduler and local coordinator shall maintain their own aggregated device and job information and use RPC for communication.

5 Evaluation

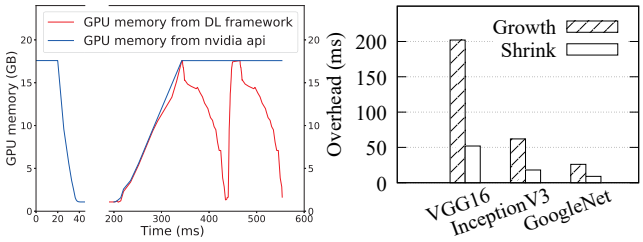
In this section, we first show micro-benchmark results to demonstrate the effectiveness and efficiency of AntMan mechanisms. We then evaluate the benefits of AntMan in a small cluster with 64 V100 GPUs to compare the policies with real workloads. Finally, we present the evaluation results on a production cluster with more than 5000 heterogeneous GPUs (V100 and P100). All the experiments are conducted on a cloud GPU cluster with 8 servers, unless explicitly stated. Every server is equipped with a 96-core Intel Xeon Platinum 8163 (Skylake) @2.50GHz with 736GB RAM, running CentOS 7.7. Each server has 8 NVIDIA V100 GPUs (32 GB GPU memory, with NVLink) powered by NVIDIA driver 418.87, CUDA 10.0, and CUDNN 7. The cloud GPU cluster is managed by Kubernetes; jobs are submitted through KubeFlow, and are executed in Docker containers. Only data-parallel is evaluated with synchronous training for jobs that require more than 1 GPU because they are common, although asynchronous training can also be supported. The trace in the experiment consists of 9 models, 2 of them implemented

	Model	Arrival	GpuMem	BS	Quota
Job-A	GCN	0 min	3.5 GB	1400	No
Job-B	ResNet	26 min	30.0 GB	360	Yes

Table 1: Setup and information of two jobs.

	Preempt	FIFO	Pack	UMem	AntMan
Job-A	Failed	43.0	43.1	43.4	43.9
Job-B	91.1	108.2	Failed	541.6	91.8

Table 2: Job status and JCT (min) of two jobs executing in different configurations.



(a) A shrink-growth profiling on ResNet-50.

(b) Overhead of GPU memory scaling for typical models.

Figure 11: Efficiency of GPU memory scaling in AntMan.

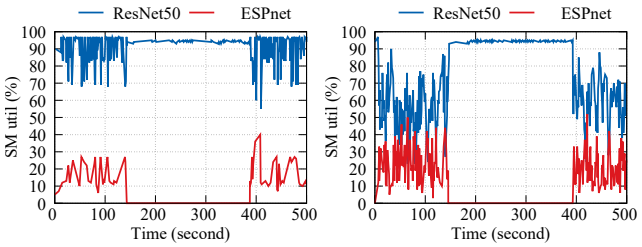
in PyTorch 1.3.1 and 7 of them implemented in TensorFlow 1.12.

5.1 Benchmark

In this section, we evaluate the dynamic scaling mechanism of AntMan in two aspects, memory and computation unit. We first demonstrate that dynamic memory scaling is indispensable in preventing failure and ensuring job performance. We then measure the efficiency of memory shrinkage and growth on typical workloads and detail the timeline on a ResNet-50 benchmark. Finally, we demonstrate the ability of dynamic computation unit scaling on avoiding job interference, by packing two jobs in a shared GPU.

Dynamic GPU memory scaling. To demonstrate that dynamic memory scaling is essential for sharing GPUs with multiple jobs, two typical jobs are chosen to construct a typical scenario. As shown in Table 1, Job-A is a GCN model that arrives at 0 minutes. Its peak GPU memory usage is 3.5 GB and is submitted by users without a resource quota. Job-B is a ResNet-50 task that arrives 26 minutes later. In total, it consumes 30 GB GPU memory and is submitted with a resource quota guarantee, which means it should run directly to meet the SLA requirements. The cluster has only one 32 GB GPU left and both jobs are scheduled on this GPU at arrival. Both jobs are run in the setup described above multiple times, but with different action policies when Job-B arrives. Table 2 shows the job status and job completion time (JCT) in minutes for both jobs with different configurations. At Job-B's arrival, the scheduler can choose to preempt Job-A. In this way, Job-B can be directly scheduled and finished in 91.1 minutes at the cost of Job-A's failure. The second choice is to run Job-B in a first-in-first-out (FIFO) mode. Job-B will not be launched until Job-A is finished, which introduces an extra 17.1-minute queuing delay. The third choice is to pack two jobs in the same GPU as proposed in Gandiva [48]. In this case, Job-B eventually fails because of the insufficient GPU memory (28.5 GB) granted. UMem indicates running Job-B in packing mode with the support of AntMan's universal memory, but without the coordinated scaling on the GPU memory limit (Section 3.1.1). Host memory are used when running out of GPU memory. Thus, Job-B will not fail from out-of-memory, however, it takes 514.6 minutes to finish and violates the SLA. AntMan leverages both universal memory and dynamic GPU memory scaling to coordinate job execution. It allocates sufficient device memory to Job-B as it runs with a resource quota, and offers the rest part of GPU memory to Job-A to allow it run as efficiently as possible. More specifically, when Job-B arrives, AntMan coordinates two jobs to shrink the GPU memory usage of Job-A and grow the GPU memory of Job-B. Job-B uses 30 GB GPU memory and Job-A uses the 2 GB left over, and 1.5 GB host memory. Note that, the performance of Job-B is still slightly slower compared to the preemptive scenario. This is because even though the required GPU memory is sufficient through dynamic scaling of AntMan, Job-B is still interfered in by the co-execution with Job-A in the computation unit.

Efficient memory shrinkage and growth. To demonstrate the efficiency of the dynamic memory scaling mechanism, a ResNet-50 job is run and the memory shrinkage and growth are manually triggered in order. As shown in Figure 11a, the performance is measured by monitoring the in-use GPU memory using both Nvidia API and memory statistics in DL frameworks. As Figure 11a indicates, the memory shrink from 17.6 GB to 1.3 GB takes only 17 *ms*. The GPU memory usage grows back to 17.6 GB in 143 *ms*, which is slower than the memory shrink. This is because GPU memory is allocated on demand with deep learning forward computation. Thus, the measured time includes both the forward computation time, which is essential to this mini-batch, and the memory allocation overhead. To understand the actual overhead, the time cost and memory usage of the next mini-batch are also plotted. The mini-batch with GPU memory growth takes 234 *ms* and the next mini-batch, which utilizes the cached memory, takes 119 *ms* to accomplish. Therefore, the growth overhead of ResNet-50 model is 115 *ms*. The same approach is applied to measure memory scaling overhead on other typical DL models. Figure 11b summarizes the overhead measured for VGG16 [43], Inception3 [45], and GoogleNet [44], which adjust GPU memory at a size of 17 GB, 16 GB, and 4 GB



(a) Packing mode.

(b) Adaptive computation adjustment mode.

Figure 12: The SM utilization rates of packing mode in Gandiva [48] and an adaptive computation adjustment mode in AntMan for a 500s segment of execution of ESPnet and ResNet-50.

respectively. Given a dynamic memory scaling interval of one minute, the largest overhead (*i.e.*, VGG16) is still negligible (only 0.4%).

Dynamic GPU computation unit scaling. To demonstrate the adaptive computation adjustment is essential for sharing GPU between multiple jobs, the SM utilization rates when running two typical jobs under packing mode and adaptive computation mode are characterized separately. As shown in Figure 12, the resource-guarantee job is an PyTorch job with ESPnet [46] model on the speech-text dataset. It co-executes with an opportunistic job which is a TensorFlow job with ResNet-50 [22] model on ImageNet [16]. Compared to ResNet-50, ESPnet consumes less SM and less memory. Therefore, packing these two jobs together into one GPU incurs a relatively higher GPU kernel queuing delay for the ESPnet and eventually leads to an SLA violation. Figure 12a illustrates that ESPnet is poor at competing GPU computation cycles compared to ResNet-50. The utilization of ESPnet remains mostly at 30% which is lower than in Figure 12b. ResNet-50 launches many more kernels per unit time than ESPnet, therefore, it consumes more GPU computation time. These results show that the end-to-end execution time of ESPnet increases dramatically from 20.1 minutes (when running on a dedicated GPU) to 105.2 minutes (when running together with ResNet-50).

Figure 12b illustrates that AntMan can leverage adaptive computation adjustment to utilize the left over resources as much as possible while still satisfying the SLA requirements. Specifically, AntMan introduces a feedback-based adjustment approach that continuously monitors the performance of resource guarantee jobs and uses performance feedbacks to adjust the GPU kernel launching frequency of opportunistic jobs. As shown in Figure 12b, the SM utilization rates of the training stage (the first 140 seconds) of ESPnet fluctuate between 5% and 50%. In this scenario, AntMan continuously adjusts the GPU kernel launching frequency of ResNet-50 to ensure the training performance of ESPnet. Therefore, the results reflected in this figure is that the SM utilization rates of ResNet-50 are constantly fluctuating between 30% to 90% within the first 140 seconds of execution. In contrast, the decoding stage (between 140 and 390 seconds) of ESPnet runs without consuming GPU computation cycles. Therefore, the SM utilization rates of ResNet-50 are relatively high at this stage As a result, by leveraging adaptive computation adjustments, the end-to-end execution time of ESPnet remains 20.8 minutes while ResNet-50 maintains 57% performance.

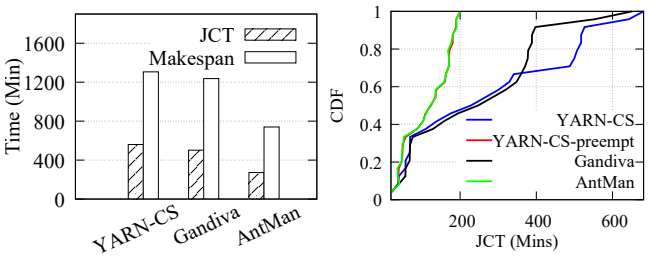
	Model	Type	Dataset
20%	ResNet-50 [22]	CV	ImageNet [16]
	VGG16 [43]	CV	Cifar10 [30]
	SuperResolution [42]	CV	BSD300 [34]
20%	Bert [17]	NLP	SQuAD [38]
20%	ESPnet [46]	Speech	Corp.Data
20%	GraphSAGE [21]	Rec.	PPI [55]
	GCN [29]	Rec.	Cora [41]
20%	DIN [53]	Ad.	Corp.Data
	Wide & Deep [15]	Ad.	Corp.Data

Table 3: Deep learning models and the ratios in the trace.

5.2 Trace Experiment

Workloads. Nine state-of-the-art deep learning models are selected from Github, together with open datasets, as summarized in Table 3. As the datasets of speech and advertisement are too small for evaluation, the internal datasets of Alibaba are used for the experiment. The models are classified into categories according to their application domains and they are evenly mixed up (20%). The job runtime of the trace is configured according to the distribution reported by Microsoft [48]. As a simplified multi-tenant setup, deep learning training jobs of the trace are randomly dispatched into two tenants. Tenant-A has 64-GPU quota and Tenant-B has no quota. Therefore, all Tenant-A's jobs are resource-guarantee jobs, and all jobs in Tenant-B are opportunistic jobs.

Baseline. The experiment compares AntMan to another GPU production cluster scheduler, Apache YARNs capacity scheduler (YARN-CS), which is used in Microsoft Philly [19, 28]. Gandiva [48], a state-of-the-art DL scheduling system, is also used for comparison. Gandiva introduces a series of primitives in DL for scheduling, including packing, migration, and time-slicing. The packing strategy of Gandiva is used in this experiment, which greedily schedules jobs to the GPUs with lowest GPU utilization and sufficient GPU memory. The migration and time-slicing proposed in Gandiva are to solve resource fragmentation and benefit AutoML, which are orthogonal to AntMan. Note that, Gandiva relies on job profiling information (*i.e.*, GPU utilization, GPU memory usage) for greedy packing decisions. Such profiling can hardly



(a) Comparison of YARN-CS, Gandiva, and AntMan. (b) Job completion time of resource-guarantee jobs.

Figure 13: Trace experiment on 64 V100 GPUs.

be achieved in a production cluster, as its outputs might affect the successor tasks of DL pipeline. In the trace experiment, profiling information is unknown to both AntMan and YARN-CS.

Results. Figure 13a shows the average job completion time (JCT) and the makespan for the three schedulers when executing the same synthesized job trace in a cluster with 64 V100 GPUs. Compared to the capacity scheduler and Gandiva, AntMan improves average JCT by 2.05x and 1.84x. The total makespan is also reduced by 1.76x and 1.67x respectively. To understand the improvements brought about by AntMan, we config YARN-CS to run with preemption, which allows jobs in Tenant-A to preempt jobs in Tenant-B for execution. The JCT of resource-guarantee jobs (Tenant-A) are shown in Figure 13b. This shows the JCT of AntMan is almost the same as YARN-CS-preempt, however, YARN-CS-preempt achieves it with 46% of jobs being preempted. AntMan respects the jobs of Tenant-A and schedules them once their resource quota are satisfied, while conducting a performance control on the co-executing opportunistic jobs to avoid interference. Conversely, Gandiva delays the completion time of these jobs because of the lack of performance isolation and dynamic resource scaling.

5.3 Cluster Experiment

AntMan has been deployed on the production clusters of Alibaba to serve tens of thousands of daily deep learning training jobs. To verify the design and implementation of AntMan while ensuring it works properly, experiments and statistics are conducted on a heterogeneous GPU cluster with over 5000 GPUs.

To illustrate the cluster efficiency improvement provided by AntMan, one-week statistics were collected in December 2019, right before the deployment of AntMan, as the baseline. It is compared to the number collected in April 2020, after AntMan was fully deployed for weeks. However, as the jobs of these two weeks are different, the average JCT cannot be compared directly. Therefore, we focus on system metrics

	Avg.	90% tile	95% tile
Dec. 2019	1132	1978	5960
Apr. 2020	550	124	489

Table 4: One-week queuing delay statistic in seconds.

Interference	0%	0∼1%	1∼2%	2∼3%	3∼4%
# of jobs	9895	26	30	20	29

Table 5: Interference analysis on mini-batch time for 10K production jobs

comparison because the jobs of this cluster come from the same departments in Alibaba. The comparison shows that AntMan provides up to 17.1% extra GPUs for DL training jobs in this cluster. Hardware statistics show that AntMan achieves a 42% improvement on average for GPU memory usage and a 34% improvement on average for GPU utilization. Table 4 illustrates the queuing delay of jobs selected from a one-week period when roughly the same number of jobs arrive at the cluster. It illustrates that on average, the job queuing delay reduces by 2.05x and the tail latency significantly reduces by more than an order of magnitude, thanks to the cluster throughput improvement.

To measure the performance of resource-guarantee jobs in co-execution, 10000 jobs were randomly sampled from one week in April 2020 which both have the phases executing exclusively and co-executing with other jobs. For each job, the mini-batch time was recorded for both its dedicated execution and packing execution with other jobs. The mini-batch time difference between these two scenarios was calculated and any gaps larger than 10 *ms* were considered as interfered (10 *ms* is small enough to be considered as mini-batch fluctuation). In this way, the interference ratio for each job could be calculated. As shown in Table 5, 99% of the jobs suffer zero performance downgrades during job packing.

6 Related Work

GPU memory management. To optimize the limited and valued GPU memory for supporting larger batch-size DNN training, vDNN [39], Capuchin [36], CDMA [40], and Gist [26] adopts eviction, prefetching, and re-computation to reduce the GPU memory footprint, leveraging application-specific knowledge. Salus [50] packs multiple jobs in the same process to share the GPU memory management, however, with interference in co-execution. In addition, running multiple jobs in a process could potentially broadcast the failures, especially when given a significantly high failure ratio [27, 51]. AntMan provides a universal memory management design using dynamic GPU and CPU memory swapping at the granularity of tensors for the fluctuant load, which complements the memory swapping and re-computation policies.

Interference and performance isolation. Performance isolation is critical in modern operating systems and shared CPU clusters. Linux uses cgroups [1] to control the CPU and memory usage of a process. However, it rarely has support for general GPU applications. A series of research works, such as Quincy [25] and Entropy [24], optimize the job performance for fair sharing on CPU clusters. In AntMan, the characteristic of DL jobs is leveraged to provide fine-grained control on GPU memory and computation unit at runtime, which is similar to cgroups, but on an application level.

The interference issue of multiplexing jobs on a GPU has been well studied. Baymax [14] shares GPUs by mitigating queuing delay and PCIe contention. Prophet [13] tries to predict co-executed GPU workload performance using an analytical model. AntMan introduces an operator management module in the executor of the DL framework, leveraging the inherent periodical mini-batch iteration cycles as a metric for inter-job coordination. It controls the frequency of GPU kernel launches and resolves the contention in both the GPU computation unit and PCIe.

NVIDIA MPS can co-operate with multi-process CUDA applications in a GPU. MPS support is not production ready yet [4]. The resource limit cannot be changed at the runtime of a client process which violates the fluctuant characteristic. Moreover, MPS merges CUDA execution in only one context, resulting in the termination of all clients for any fatal GPU exceptions. rCUDA [37] and FlexDirect [3] of VMWare Bitfusion allow jobs to be remotely executed on a shared GPU.

GPU cluster scheduling Today, DL training jobs in multi-tenant production clusters are managed by infrastructures such as Kubernetes or YARN [9,28], where jobs are allocated on dedicated GPUs, leading to common low utilization [27]. Gandiva [48] proposes time-slicing, migration, and packing to allow GPU sharing. Time-slicing and migration switch the GPU usage among jobs in coarse-grained, and therefore cannot improve GPU utilization. The packing approach proposed in Gandiva [48] could potentially introduce significant unpredictable resource contention, which violates the fairness requirements of a shared multi-tenant cluster. Themis [33] addresses the unfairness of placement-sensitive characteristic in DL jobs by proposing a long term fairness object. Gandiva$_{fair}$ [12] addresses the fairness issue of multi-size job time-slicing and proposes an automated trading mechanism. AlloX [31] efficiently and fairly schedules DL jobs in interchangeable resources by modelling the scheduling problem as a min-cost bipartite matching problem. AntMan introduces opportunistic DL jobs as low-priority jobs to best-effort utilize the GPU cycles, which is complementary to the fairness metrics and policies proposed above.

Elastic training. To utilize the idle GPUs introduced by gang-scheduling and to support fault-tolerance in DL training, TorchElastic [7] and ElasticDL [2] are designed to start training with any number of available GPUs. A common problem of these elastic DL frameworks is that the model training accuracy can hardly be guaranteed or reproduced, and are thus rarely used in production.

7 Conclusion

We present AntMan, a deep learning infrastructure deployed in the GPU production clusters of Alibaba. AntMan introduces dynamic scaling primitives in deep learning frameworks, allowing flexible fine-grained control of GPU resources for individual deep learning jobs at runtime. By utilizing the effective primitives mentioned above, AntMan co-designs cluster scheduler and deep learning frameworks for cooperative job management, allowing GPUs to be utilized by over-provision of opportunistic jobs at best-effort while avoiding the interference to other jobs. AntMan improves the overall GPU memory utilization and the computation unit utilization of Alibaba's GPU clusters by 42% and 34% respectively without compromising fairness.

Acknowledgements

We would like to thank our shepherd Roxana Geambasu and the anonymous reviewers for their valuable comments and suggestions. We would also like to thank Chen Xing, Jin Ouyang, Xinyuan Li, Lixue Xia for their help in improving quality of writing.

References

[1] cgroups. https://en.wikipedia.org/wiki/Cgroups.

[2] ElasticDL. https://github.com/sql-machine-learning/elasticdl/.

[3] FlexDirect of VMware BitFusion. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/whitepaper/vmw-bitfusion-docs-flexdirect-whitepaper.pdf.

[4] MPS. https://github.com/NVIDIA/nvidia-docker/issues/419.

[5] NCCL. https://developer.nvidia.com/nccl/.

[6] NVLink. https://www.nvidia.com/en-us/data-center/nvlink/.

[7] TorchElastic. https://github.com/pytorch/elastic.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, volume 16, pages 265–283. USENIX Association, 2016.

[9] Scott Boag, Parijat Dube, Benjamin Herta, Waldemar Hummer, Vatche Ishakian, K Jayaram, Michael Kalantar, Vinod Muthusamy, Priya Nagpurkar, and Florian Rosenberg. Scalable multi-framework multi-tenant lifecycle management of deep learning training jobs. In *Workshop on ML Systems, NIPS*, 2017.

[10] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association.

[11] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14:70–93, 2016.

[12] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[13] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32, 2017.

[14] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 681–696. ACM, 2016.

[15] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[18] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[19] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[20] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: A factorization-machine based neural network for CTR prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1725–1731. ijcai.org, 2017.

[21] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 1024–1034, 2017.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9, 2014.

[24] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009*, pages 41–50. ACM, 2009.

[25] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[26] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE, 2018.

[27] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[28] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Tech. Rep.*, 2018.

[29] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[30] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[31] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[32] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. Plink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training, 2020.

[33] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[34] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision*, volume 2, pages 416–423, July 2001.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[36] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[37] Javier Prades and Federico Silla. Gpu-job migration: The rcuda case. *IEEE Trans. Parallel Distrib. Syst.*, 30(12):2718–2729, 2019.

[38] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2383–2392. The Association for Computational Linguistics, 2016.

[39] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 18:1–18:13. IEEE Computer Society, 2016.

[40] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

[41] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Mag.*, 29(3):93–106, 2008.

[42] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video

super-resolution using an efficient sub-pixel convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1874–1883, 2016.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[44] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[45] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[46] Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. Espnet: End-to-end speech processing toolkit. In *Interspeech*, pages 2207–2211, 2018.

[47] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter R. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, pages 84–97. ACM, 2016.

[48] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[49] Wencong Xiao, Zhenhua Han, Hanyu Zhao, Xuan Peng, Quanlu Zhang, Fan Yang, and Lidong Zhou. Scheduling CPU for gpu-based deep learning jobs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, page 503. ACM, 2018.

[50] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.

[51] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An empirical study on program failures of deep learning jobs. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20, pages 1159–1170, NY, USA, 2020. Association for Computing Machinery.

[52] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of the VLDB Endowment*, volume 7, pages 1393–1404. VLDB Endowment Inc., 2014.

[53] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1059–1068, 2018.

[54] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019.

[55] Marinka Zitnik and Jure Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198, 2017.

[56] Barret Zoph, Ekin D. Cubuk, Golnaz Ghiasi, Tsung-Yi Lin, Jonathon Shlens, and Quoc V. Le. Learning data augmentation strategies for object detection. *CoRR*, abs/1906.11172, 2019.

Write Dependency Disentanglement with HORAE

Xiaojian Liao, Youyou Lu*, Erci Xu, and Jiwu Shu*

Tsinghua University

Abstract

Storage systems rely on write dependency to achieve atomicity and consistency. However, enforcing write dependency comes at the expense of performance; it concatenates multiple hardware queues into a single logical queue, disables the concurrency of flash storage and serializes the access to isolated devices. Such serialization prevents the storage system from taking full advantage of high-performance drives (e.g., NVMe SSD) and storage arrays.

In this paper, we propose a new IO stack called HORAE to alleviate the write dependency overhead for high-performance drives. HORAE separates the dependency control from the data flow, and uses a dedicated interface to maintain the write dependency. Further, HORAE introduces the joint flush to enable parallel `FLUSH` commands on individual devices, and write redirection to handle dependency loops and parallelize in-place updates. We implement HORAE in Linux kernel and demonstrate its effectiveness through a wide variety of workloads. Evaluations show HORAE brings up to 1.8× and 2.1× performance gain in MySQL and BlueStore, respectively.

1 Introduction

The storage system has been under constant and fast evolution in recent years. At the device level, high-performance drives, such as NVMe SSD [15], are pushed onto the market with around 5× higher bandwidth and 6× lower latency against their previous generation (e.g., SATA SSD) [11, 18]. From the system perspective, developers are also proposing new ways of storage array organization to boost performance. For example, in BlueStore [23], a state-of-the-art storage backend of Ceph [45], data, metadata and journal can be separately persisted in different, or even dedicated devices.

With drastic changes from the hardware to the software, maintaining the write dependency without severely impacting the performance becomes increasingly challenging. The write dependency indicates a certain order of data blocks to be persisted in the storage medium, and further underlies a variety of techniques (e.g., journaling [44], database transaction [13]) to provide ordering guarantee in the IO stack. Yet, the write order is achieved through an expensive approach, referred as *exclusive IO processing* in this paper. In the exclusive IO processing, the following IO requests can not be processed until the preceding one has been transferred through PCIe bus, then been processed by the device controller and finally returned with a completion response.

Unfortunately, this one-IO-at-a-time fashion of processing conflicts with the high parallelism of the NVMe stack, and further nullifies the concurrency potentials between multiple devices. First, it concatenates the multiple hardware queues of the NVMe SSD, thereby eliminating the concurrent processing of both host- and device-side cores [50]. Moreover, it serializes the access to physically independent drives, preventing the applications from enjoying the benefits of aggregated devices. In our motivation study, we observe that with the scaling of hardware queues and devices, the performance loss introduced by the write dependency can be up to 87%. Conversely, orderless writes without dependency can easily saturate the high bandwidth of NVMe SSDs (§3).

Therefore, to leverage the high bandwidth of NVMe SSDs while preserving dependency, we propose the *barrier translation* (§4) to convert the ordered writes into orderless data blocks and ordering metadata that describes the write dependency. The key idea of barrier translation is shifting the write dependency maintenance to the ordering metadata during normal execution and crash recovery, while concurrently dispatching the orderless data blocks.

We incarnate this idea by re-architecting modern IO stack with HORAE (§5). In a nutshell, HORAE bifurcates the traditional IO path into two dedicated ones, namely ordered control path and orderless data path. In the control path, HORAE flushes ordering metadata directly into the devices' persistent controller memory buffer (CMB), a region of general-purpose read/write memory on the controller of NVMe SSDs [15, 16], using memory-mapped IO (MMIO). On the other hand, HORAE reuses classic IO stack (i.e., block layer to device driver to

*Jiwu Shu and Youyou Lu are the corresponding authors.
{shujw, luyouyou}@tsinghua.edu.cn

device) to persist orderless writes. Note that this design is also scaling-friendly as orderless data blocks can be processed in both an asynchronous (to a single device) and pipelined (to multiple devices) manner.

Now, with a bifurcated IO path, we further develop a series of techniques to ensure both high performance and consistency in HORAE. First, we design *compact ordering metadata* and efficiently organize them in the CMB (§5.2). Second, the *joint flush* of HORAE performs parallel FLUSH commands on dependent devices (§5.3). Next, HORAE uses the write redirection to break the dependency loops, *parallelizing in-place updates* with strong consistency guarantee (§5.4). Finally, for crash recovery, HORAE reloads the ordering metadata, and further only commits the valid data blocks but discarding invalid ones that violate the write order (§5.5).

To quantify the benefits of HORAE, we build a kernel file system called HORAEFS to test applications relying on POSIX interfaces, and a user-space object store called HORAESTORE for distributed storage (§5.6). We compare HORAEFS against ext4 and BarrierFS [46], resulting in an up to 2.5× and 1.8× speedup at file system and application (e.g., MySQL [13]) level, respectively (§6). We also evaluate HORAESTORE against BlueStore [23], showing the transaction processing performance increases by up to 2.1×.

To sum up, we make the following contributions:

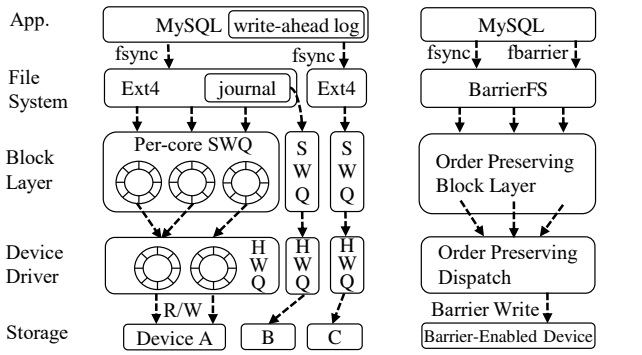
- We perform a study of the write dependency issue on multi-queue drives among both single and multiple devices setup. The results demonstrate considerable overhead of ensuring write dependency.
- We propose the barrier translation to decouple the ordering metadata from the dependent writes to enforce correct write order.
- We present a new IO stack called HORAE to disentangle the write dependency of both a single physical device and storage arrays. It introduces a dedicated path to control the write order, and uses joint flush and write redirection to ensure high performance and consistency.
- We adapt a kernel file system and a user-space object store to HORAE, and conduct a wide variety of experiments, showing significant performance improvement.

2 Background

This section starts with a brief introduction of enforcing write dependency under current IO stack (§2.1). Then, we illustrate state-of-the-art techniques that alleviate the overhead of enforcing the write dependency (§2.2).

2.1 Basic Ordering Guarantee Approach

In Figure 1(a), we can see that modern IO stack is a combination of software (i.e., the block layer, the device driver) and hardware (i.e., the storage device) layers. Each layer may reorder the write requests for better performance [15, 50] or fairness [4, 29]. Specifically, in the block layer, the host IO scheduler can schedule the requests in the per-core software queues based on different algorithms (e.g., deadline). While in the storage device, the controller may fetch and process arbitrary requests due to timeouts and retries.



(a) Linux IO stack **(b)** Barrier-Enabled IO stack

Figure 1: Existing IO Stacks with Different Order-Preserving Techniques. *SWQ: software queue. In current multi-queue block layer, each core has a software queue. HWQ: hardware queue. The storage device determines the maximum number of HWQs. Emerging NVMe drives usually have multiple HWQs.*

As a result of this design, the file system must explicitly enforce the storage order. Traditionally, the file system relies on two important steps: *synchronous transfer* and *cache barrier* (e.g., cache FLUSH). First, synchronous transfer requires the file system to process and transfer the dependent data blocks through each layer to the storage interface serially. Then, to further avoid the write reordering by the controller in the storage device layer, the file system issues a cache barrier command (e.g., FLUSH), draining out the data blocks in the volatile embedded buffer to the persistent storage. Afterwards, the file system repeats this processing the next request. Through interleaving dependent requests with exclusive IO processing, the file system ensures the write requests are made durable with the correct order.

The basic approach in guaranteeing the storage order is undoubtedly expensive. It exposes DMA transfer latency (synchronous transfer) and flash block programming delay (cache barrier) to the file system.

2.2 Ordering Guarantee Acceleration

Many techniques [25, 26, 46, 48] improve the basic approach presented in §2.1, by reducing the overhead of synchronous transfer and cache barrier. As barrier-enabled IO stack (BarrierIO [46]) is the closest competitor, we introduce it briefly.

BarrierIO reduces the storage order overhead by preserving the order throughout the entire IO stack. Specifically, the BarrierIO enforces the write dependency mainly using two techniques: the *order-preserving dispatch* to accelerate the synchronous transfer, and the *barrier write* command to improve the cache barrier. First, as shown in Figure 1(b), the order-preserving block layer ensures that the IO scheduler follows the write order specified by the file system. Further, the order-preserving dispatch maintains the write order of

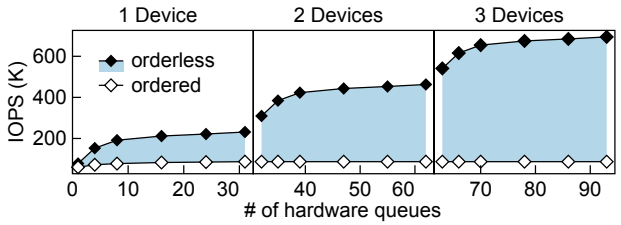


Figure 2: Ordered Write VS. Orderless Write with Varying The Number of Hardware Queues and Devices. *Each device has up to 32 hardware queues. Described in Section 3.1.*

requests queueing in the (single) hardware queue. As a collaborator, the storage controller also fetches and serves the requests in a serialized fashion. Second, BarrierIO replaces the expensive `FLUSH` with a lightweight barrier write command for the storage controller to preserve the write order.

File systems provide the `fsync()` call for applications to order their write requests. Yet, it is still too expensive to preserve order by `fsync()`. Thus, like OptFS [26], BarrierIO separates the ordering from the durability, and further introduces a file system interface, i.e., `fbarrier()`, for ordering guarantee. The `fbarrier()` writes the associated data blocks in order, but returns without durability assurance.

3 Motivation

Multi-queue. The improvement of storage technologies has been continuously pushing forward the performance of solid-state drives. To meet the large internal parallelism of flash storage, SSDs are often equipped with multiple embedded processors and multiple hardware queues [33, 38]. In the host side, as shown in Figure 1(a), the IO stack employs the multicore friendly design. It statically maps the per-core software queues to the hardware queues for concurrent access.
Multi-device. On the other hand, for higher volume capacity, performance and isolation, applications usually stripe data blocks to multiple devices as in RAID 0, or manually isolate different types of data into multiple devices. For example, as shown in Figure 1(a), the ext4 file system uses a dedicated device for journaling processing. The MySQL database redirects its write-ahead logs to a logging device.

The multi-queue and multi-device bring opportunities to enhance performance for independent write requests. Nevertheless, it still remains unknown how much overhead the write dependency introduces to the multi-queue and multi-device design. Here, we first start with a performance study of the write dependency atop multi-queue and multi-device.

3.1 Write Dependency Overhead

In this subsection, we quantify the overhead of write dependency atop Linux IO stack by comparing the IOPS of ordered writes with orderless ones. We use an NVMe SSD (spec in Table 2 Intel 750) with up to 32 hardware queues and use FIO [9] for testing. During the test, we vary the number of hardware queues and attach more devices. Further, we increase the number of threads issuing 4 KB writes to gain the maximum IOPS.

For orderless random write, we use libaio [3] engine with iodepth of 512. In this setup, the write requests issued by libaio have no ordering constraints and can be freely reordered by the storage controller according to NVMe specification [15]. The results are shown in Figure 2. As we enable more hardware queues, the IOPS of orderless writes increases and gradually saturates the storage devices.

For ordered random write, we use libaio engine but set the iodepth to 1. This setup follows the principle of exclusive IO processing in guaranteeing the ordering. As shown in Figure 2, the IOPS of ordered write can hardly grow, even if we use more devices as RAID 0 to serve the write request.

The gap between orderless and ordered writes (blue area in Figure 2) indicates the overhead of the write dependency. With the increase of hardware queues, the overhead becomes more severe, and reaches up to 87%. We conclude that the Linux IO stack is not efficient in handling ordered writes.

3.2 Write Dependency Analysis

In this subsection, we analyze the write dependency overhead via explaining the behaviors of Linux IO stack.

For a single device, the physically independent hardware queues get logically dependent. The application thread firstly puts the write requests in the software queues through the IO interface (e.g., `io_submit()`). Linux IO stack supports various IO schedulers (e.g., BFQ [4]), which perform requests merging/reordering in the software queues. Next, the requests are dispatched to hardware queues, where IO commands are generated. In the hardware queues, out of consideration for hardware performance (e.g., device-side scheduling) and complexity (e.g., request retries), there are no ordering constraints of storage controller processing the commands [15] [1]. Therefore, due to the orderless feature of both types of queues, to guarantee storage order, the IO stack only processes a single request or a set of independent requests at a time. As a result, the ordered write request keeps most queues idle and leaves the multi-queue drives underutilized.

For multiple devices, physically isolated devices get logically connected. The IO stack employs isolated in-memory data structures for the software environment of multi-device. In the hardware side, a device has its private DMA engine and hardware context. Despite the concurrent execution environment, the ordered writes flow through the multi-device in a serialized fashion; the application can not send a request to a different device until the on-going device finishes execution.

[1] In barrier-enabled devices, the host can queue multiple ordered writes and insert a barrier command in between. Such barrier command is available in a few eMMC products [22] with usually single hardware queue. Multi-queue-based NVMe does not have similar concept.

3.3 Limitation of Existing Work

A straightforward solution to remedy aforementioned issues is to keep the entire IO stack ordered as BarrierIO does, so as to allow more ordered writes to stay in the hardware queue. Recall that in Section 2.2, each layer in BarrierIO stack must preserve the request order spread by the upper layer. Implementing such design is quite simple in old drives with a sole command queue (e.g., mobile UFS, SATA SSD): the requests in the hardware queue are serviced in the FIFO way. However, it is quite challenging to extend this idea to multi-queue drives and multiple devices, without sacrificing the multicore designs of the host IO stack and the core/data parallelism of fast storage devices. We explain the reason in detail.

The key of BarrierIO is that each layer agrees on a specific order. While a single hardware queue structure itself can describe the order, for multi-queue and multi-device, the host IO stack must manually specify the order. Maintaining a global order among multiple queues throughout the entire IO stack potentially ruins the multi-queue and multi-device concurrency, which are vital features to fully exploit the bandwidth of high-performance drives, according to our evaluation (§6.2) and a recent study [50]. Further, the firmware design should comply with the host-specified order, which may introduce synchronization among embedded cores and may neutralize the internal core and data parallelism.

In this paper, instead of keeping the IO stack ordered, we seek a new approach that keeps most parts of current IO stack orderless while preserving correct storage order. We now present our design in the following sections.

4 The HORAE Foundation

To efficiently utilize the modern fast storage devices, we wish to keep the orderless and independent property of both the software and hardware intact. We achieve this goal via *barrier translation*, which disentangles the write dependency from original slow and ordered write requests.

4.1 Design

Here, we refer a series of ordered write requests issued by the file system or applications as a write stream. Commonly, a write stream can have multiple sets of data blocks and the inbetween barriers that serve as ordering points between two sets of data blocks. We note a set of data blocks to device A with a monotonic set ID x as A_x, and refer a barrier (write dependency) as $\preceq$. Thus, for a write stream $A_x \preceq B_{x+1}$, it shall be ensured that the data blocks of A_x must be made durable prior to ($\prec$) B_{x+1} or at the same time (=) as B_{x+1}.

Next, we move on to remove the dependency (i.e., $\preceq$) between two write requests. We use {} to group a set of independent write requests that can be processed concurrently. Our key issue is to translate the $A_x \preceq B_{x+1}$ into $\{A_x, B_{x+1}\}$. We decouple the indexing of a write stream from its data content. The indexing (i.e., ordering metadata) keeps a minimum set of information to retain the dependency. We refer the ordering metadata as $\tilde{A_x}$ and the data content as $\bar{A_x}$, and thus we have $A_x = \tilde{A_x} \cup \bar{A_x}$. Specifically, if A_x has consecutive data blocks of n length to device A from logical block address (lba) m, $\tilde{A_x}$ = {A, m, n}.

Given a write stream $A_x \preceq B_{x+1}$, the *barrier translation* turns it into $\tilde{A_x} \preceq \tilde{B_{x+1}} \preceq \{\bar{A_x}, \bar{B_{x+1}}\}$. Specifically, the translated write stream guarantees the order in two steps: (1) $\{\tilde{A_x}, \tilde{B_{x+1}}\} \preceq \{\bar{A_x}, \bar{B_{x+1}}\}$ and (2) $\tilde{A_x} \preceq \tilde{B_{x+1}}$. First, we must ensure the ordering metadata is made durable no later than data content. Then, the write dependency of original write stream is extracted as the ordering metadata.

4.2 Proof

Now, we further discuss the correctness of barrier translation via the following proof. Our main point is to show that the translated write stream has the same effect on satisfying the ordering constraints. In other words, the following proves that if the ordering metadata is made durable in specific order, and is made durable no later than the data content, the write dependency of original write stream is maintained.

We formalize the implication as follows:

$$\tilde{A_x} \preceq \tilde{B_{x+1}} \preceq \{\bar{A_x}, \bar{B_{x+1}}\} \implies A_x \preceq B_{x+1} \quad (1)$$

The key lies in the non-deterministic order between $\bar{A_x}$ and $\bar{B_{x+1}}$. We thus discuss the proof in two situations: $\bar{A_x} \preceq \bar{B_{x+1}}$ and $\bar{B_{x+1}} \prec \bar{A_x}$ as follows.

The first case $\bar{A_x} \preceq \bar{B_{x+1}}$. Since we already have $\tilde{A_x} \preceq \tilde{B_{x+1}}$, we have $(\tilde{A_x} \cup \bar{A_x}) \preceq (\tilde{B_{x+1}} \cup \bar{B_{x+1}})$. Because $A_x = \tilde{A_x} \cup \bar{A_x}$ and $B_{x+1} = \tilde{B_{x+1}} \cup \bar{B_{x+1}}$, we have $A_x \preceq B_{x+1}$.

The second case $\bar{B_{x+1}} \prec \bar{A_x}$. This case at first glance may seem to violate the write order. However, since we have $\tilde{B_{x+1}} \preceq \bar{B_{x+1}}$, we can always find and discard the content $\bar{B_{x+1}}$ via the indexing $\tilde{B_{x+1}}$. In this situation, the data content remains empty, i.e., $\emptyset$. The empty result obeys any write dependency, i.e., $\emptyset \in \{A_x \preceq B_{x+1}\} = \{\emptyset, \{A_x\}, \{A_x, B_{x+1}\}\}$.

Two intuitive concerns may be raised from the second case. First, a long write stream may be at a higher risk of losing more data due to discard, although it keeps a consistent state of disk status. However, such scenario is common and acceptable in storage systems. Similar to roll-back and undo log, the discarding time window (e.g., `fsync()` delay) is determined by the file systems or applications. If applications desire data durability rather than ordering, they must synchronously wait for the durability of the write stream, e.g., calling `fsync()`.

Second, a special case of the write dependency, called discarding at a dependency loop, may erase the old but valid data content. For example, consider $A_x \preceq B_{x+1} \preceq A_{x+2}$, where A_x and A_{x+2} operate on the same logical block address. This would occur when storage systems perform in-place updates (IPU). Simply discarding $\bar{A_x}$ in the second case loses the valid data. Our solution is to break the dependency loop by redirecting IPU to another location, i.e., $A_x \preceq B_{x+1} \preceq A'_{x+2}$, where A'_{x+2} targets on a different address from A_x. In this way, we guarantee the correctness of dependency loop as in

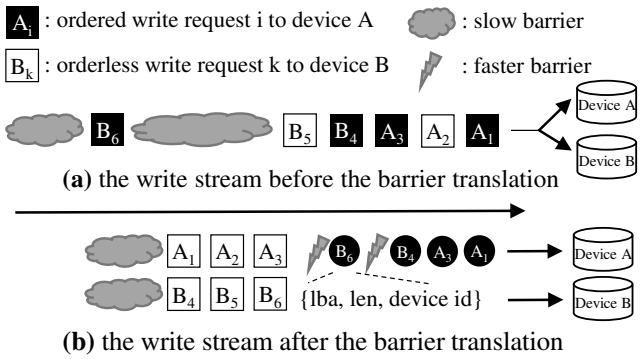


Figure 3: An Example of The Barrier Translation. *The solid circle represents the ordering metadata. The subscript denotes the desired write order of the file system or applications.*

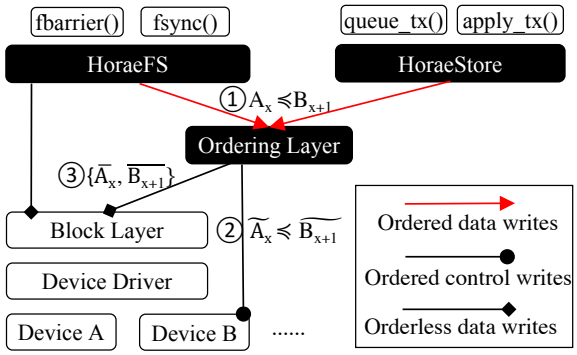


Figure 4: The HORAE IO Stack Architecture. HORAE *splits the traditional IO path into ordered control path* ② *and orderless data path* ③. HORAE *persists the ordering metadata* $\tilde{A}_x$ *via the control path before submitting the orderless data blocks* $\bar{A}_x$ *to the data path.*

normal case, preserving high concurrency as well as the old version of data. We show more details in §5.4.

4.3 Example

We now use an example to go through the entire process of barrier translation. Figure 3(a) presents the original write stream. As we can see, even for different devices, the block-sized (e.g., 4 KB) data is still processed in a serialized fashion. Even worse, the classic approach uses slow and coarse-grained barrier (the cloud shape, e.g., flash FLUSH) to enforce the relationship $\preceq$, which may process unrelated data blocks (e.g., the orderless data blocks).

Figure 3(b) shows the output of the barrier translation. Original expensive barriers are replaced with faster and fine-grained barriers (the thunder shape, e.g., memory barrier). Further, the initial ordered writes are translated to orderless ones with no barriers. As a result, after processing the ordering metadata, the devices can process data blocks at their full throttle concurrently.

5 The HORAE IO stack

To demonstrate the advantage of the barrier translation, we implement HORAE by modifying the classic Linux IO stack. The following sections first give an overview of HORAE, and then present the techniques at length.

5.1 Overview

As the high level architecture shown in the Figure 4, HORAE extends the generic IO stack with an *ordering layer* targeting ordered writes. Moreover, HORAE separates the ordering control from the data flow: the ordering layer translates the ordered data writes (①) into ordering metadata (②) plus the orderless data writes (③). The ordering metadata passes through a dedicated control path (MMIO). The orderless data writes flow through the original block layer and device driver (block IO) as usual. As a result of separation, the data path is no longer bounded by the write dependency, and it thus allows the file systems to dispatch the data blocks to arbitrary hardware queues or storage devices without exclusively occupying the hardware resources.

The key of HORAE is the ordering layer, to which the ordering guarantee of the entire IO stack is completely delegated (§5.2). Note that the ordering layer does not need to handle all block IOs, but instead just need to capture the write dependencies of ordered writes. Specifically, HORAE stores the ordering metadata in the persistent controller memory buffer (CMB in NVMe spec 1.2 [15], PMR in NVMe spec 1.4 [16]) of the storage device using an ordering queue structure. HORAE leverages epochs to group a set of writes with no intra-dependency, and further uses the ordering queue structure itself to reflect the order of each epoch with inter-dependency.

Separating the ordering control path from the data path provides numerous benefits; it saves the block layer, the device driver, and the devices from enforcing write order, which can sacrifice performance or particular property (e.g., scheduling). Further, this design enables HORAE to perform parallel FLUSHes despite the dependencies among multiple devices (§5.3). Yet, it also faces a challenge, the dependency loop.

As we mentioned in §4.2, the dependency loop occurs when multiple in-place updates (IPU) operate on the same address. As the data path of HORAE is totally orderless, multiple ordered in-progress (issued by the file system but the completion response is not returned) IPUs can co-exist and be freely reordered. The dependency loops, if not properly handled, can introduce data version issue (e.g., the former request overwrites the later one) and even the security issue (e.g., unauthorized data access). HORAE breaks the dependency loops by write redirection (§5.4). In other words, HORAE treats the IPUs as versioned writes, stores their ordering metadata serially, and concurrently redirects them to a pre-reserved disk location. In background, HORAE writes the redirected data blocks back to their original destination. In this way, HORAE parallels the IPUs while retaining their ordering.

Atop the ordering layer, HORAE exports block device abstraction and provides ordering control interface to the upper

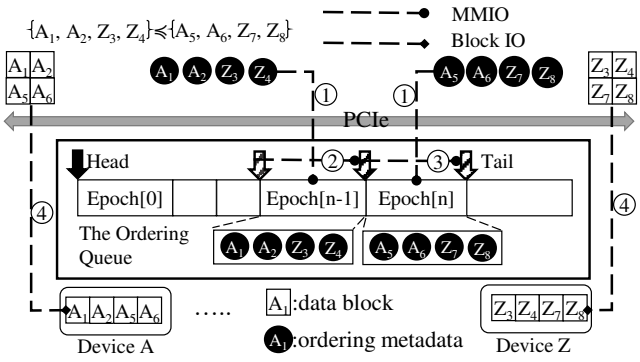


Figure 5: The Circular Ordering Queue Organization. *The ordering queue is located in the persistent controller memory buffer (CMB) of SSD.* Horae *groups a set of independent writes to an epoch, and enforces the ordering between epochs. Described in Section 5.2.*

layer systems (§5.6). We provide APIs (application programming interfaces) and high level porting guidelines for upper layer systems to adapt to Horae. Moreover, we build a kernel file system, HoraeFS, for applications that rely on POSIX file system interfaces, and a user-space object store, HoraeStore, for distributed storage backend.

On top of HoraeFS and HoraeStore, similar to previous works [23,26,46], we provide two interfaces, the ordering and durability interface, for upper layer systems or applications to organize the ordered data flow. The ordering interfaces (i.e., `fbarrier()`, `queue_tx()`) send the data blocks to the storage with ordering guarantee, but return without ensuring durability. The durability interfaces (i.e., `fsync()`, `apply_tx()`) deliver the intact semantics as before.

5.2 Ordering Guarantee

The major role of the ordering layer is guaranteeing the write order. Recall that the write stream $A_x \preceq B_{x+1}$ is translated to $\tilde{A_x} \preceq \tilde{B_{x+1}} \preceq \{\bar{A_x}, \bar{B_{x+1}}\}$, thus the ordering layer enforces the write dependency of the ordering metadata before dispatching the translated data blocks. We first present the organization of the ordering metadata.

Ordering metadata organization. As shown in the center of Figure 5, through the PCIe base address register, Horae uses the persistent Controller Memory Buffer (CMB) as the persistent circular ordering queue. The ordering queue is bounded by the `head` and `tail` pointers, and stores the ordering metadata of one write stream. For an incoming ordered write request, Horae first appends its ordering metadata to the queue via MMIO. As shown in Figure 6, the ordering metadata is compact, mainly consisting of range-based destination (i.e., `lba`, `len`, `devid`). Storing the ordering metadata via control path is a CPU-to-device transfer with byte-addressability; unlike an interrupt-based memory-to-device DMA transfer, it does not transfer a full block nor switch context. Thus, persisting the compact ordering metadata via MMIO is efficient.

Horae leverages the epoch to group a set of independent writes. And, Horae only enforces the write dependency in the unit of epoch. To realize epoch, Horae uses the `etag` to indicate the boundary of epochs. The `etag` implies $\preceq$.

Format:	lba	len	devid	etag	dr	plba	rsvd
Size in bits:	32	32	8	1	1	32	22

Figure 6: The Ordering Metadata Format. *lba: logical block address. len: length of continuous data blocks. devid: destination device ID. etag: epoch boundary. dr: is made durable. plba: lba of prepare write. rsvd: reserved bits.*

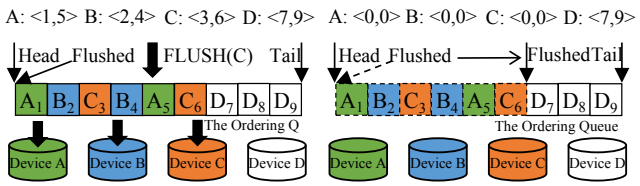
Now, we go through an example presented in Figure 5 to show how data blocks reach the storage with ordering constraints. Suppose that the ordering layer receives two sets of ordered write requests from two threads with ordering constraints $\{A_1,A_2,Z_3,Z_4\} \preceq \{A_5,A_6,Z_7,Z_8\}$. The ordering layer first forms two epochs `N` and `N-1`, and constructs the ordering metadata according to the data blocks and write dependencies. Next, the two threads store the ordering metadata concurrently to the ordering queue via MMIO (①). Then, Horae updates the 8-byte `tail` pointer sequentially to ensure both the update atomicity of each epoch and the write order of associated ordering metadata (②, ③). Finally, the two threads dispatch the orderless writes concurrently via block IO interface (④).

Since the size of available CMB is usually limited (e.g., 2 MB), the ordering queue may exceed the CMB region. Thus, Horae introduces two operations, swap and dequeue, to reclaim free space of the CMB for incoming requests.

Swap. The ordering layer blocks the threads issuing ordered writes, and then invokes swap operation, when the total size of the valid ordering metadata exceeds the queue capacity. It moves the valid ordering metadata to a checkpoint region with larger capacity, e.g., a portion of flash storage. It first waits for the on-going append operations on the ordering queue to complete. Next, it copies the whole valid ordering metadata to the checkpoint region. Finally, it updates the `head` and `tail` pointers atomically with a lightweight journal.

Dequeue. The ordering layer dequeues the expired ordering metadata when its associated data blocks and the preceding ones are durable. The dequeue operation moves the 8-byte `head` pointer.

Optimization. We observe that the MMIO write latency through PCIe is acceptable, but MMIO read can be extremely slow (8 B read costs 0.9us, 4 KB read costs 113us). This is because MMIO read is split into small-sized (determined by CPU) non-posted read transactions to guarantee atomicity [17]. Yet, MMIO reads can be abundant on the CMB. For example, Horae allocates free locations from the ordering queue before sending the ordering metadata, which requires frequent `head` and `tail` pointer access. Also, Horae needs to read the whole ordering queue for a swap operation. Horae avoids slow MMIO reads by maintaining an in-memory write-through cache for the entire CMB. The cache serves all read operations in memory. Write operations are performed to the cache, and persisted to the device CMB simultaneously via



(a) Pre-FLUSH states **(b)** Post-FLUSH states

Figure 7: The Durability and Joint FLUSH of HORAE. *A FLUSH command to a single device also triggers flushings to other devices whose write requests must be made durable in advance.*

MMIO. The extra memory consumption (2 MB) is negligible.

5.3 Durability Guarantee

Applications may also require instant durability. Linux IO stack uses the FLUSH command (e.g., calling `fsync()`) to enforce durability and ordering of updates in a storage device. Further, the FLUSH serves as a barrier between multiple devices, so as to ensure that the post-FLUSH requests are not made durable prior to the pre-FLUSH ones. For example, to guarantee the durability of $A_x \preceq B_{x+1}$, Linux IO stack issues two FLUSHes. The first one on device A ensures the write dependency ($\preceq$) as well as the durability of A_x, and the second one on device B for durability of B_{x+1}.

The FLUSH of HORAE no longer serves as a barrier. Thus, HORAE eliminates intermediate FLUSHes and invokes an eventual FLUSH to ensure durability. Unlike the legacy FLUSH targeting on sole device, the FLUSH of HORAE, called joint FLUSH, automatically flushes related devices whose data blocks should be persisted in advance.

Figure 7 shows an example of the joint FLUSH in detail. The states of the ordering queue are in Figure 7a, and suppose we decide to flush device C. The ordering layer firstly finds the flush point (6), the last position of the flushed device in the ordering queue. Next, it identifies the devices that need to be flushed simultaneously. A device should be flushed if it has requests prior to the flush point. To quickly locate the flush candidates, HORAE keeps the first and last position of each device (device range) in the ordering queue (e.g., ⟨1,5⟩ of device A), as shown in the top of Figure 7a. By checking the existence of the intersection of the device range and head-to-flush range (⟨1,6⟩), HORAE selects the flush candidates. Then, HORAE sends FLUSH commands to the candidate devices (A and B) simultaneously. When the joint FLUSH completes, HORAE moves the `flushed` pointer and resets the device range, as shown in Figure 7b.

With the `flushed` pointer, HORAE ensures the durability of the write requests between the `head` and `flushed` position. However, on SSDs with power loss protection (PLP), data blocks are guaranteed to durable when they reach the storage buffer, prior to a FLUSH command. Therefore, HORAE uses the `dr` bit (shown in Figure 6) to indicate the durability of the requests after the `flushed` position. For SSD with PLP, HORAE sets the `dr` bit, once the ordered write requests are completed via interrupt handler or polling.

While the legacy FLUSH is always performed in a serialized and synchronous fashion, the joint FLUSH enables HORAE to flush concurrently and asynchronously, for devices without PLP. These devices expose extremely long flash programming latency, so async flushings on them exploit the potential parallelism. However, HORAE remains the sync flushing on the other type of devices with PLP. Since flushing such devices is returned from the block layer directly, async flushing incurs unnecessary context switches from the wakeup mechanism.

5.4 Handling Dependency Loops

To understand the motivation for resolving the dependency loops, we show two examples. First, consider a data block is overwritten repeatedly. Reordering of two overwrite operations may cause the later one to be overwritten by the former one, and results in a data version issue. Second, consider at the file system level. The file system frees a data block from owner A, and then reallocates it to owner B. Reordering of reallocating and freeing upon a sudden crash can cause a security issue: owner B can see the data content of owner A.

Classic IO stacks handle dependency loops by prohibiting multiple in-progress IPUs on the same address. IPUs on the same address are operated exclusively, where the next IPU can not be submitted until the preceding one is completely durable. However, this approach serializes the access to the same address, leaving the device underutilized. HORAE allows multiple in-progress parallel IPUs on the same address, and resolves dependency loops through IPU detection, prepare and commit write.

IPU detection. The foremost issue is to detect IPUs. In HORAE, the upper layer systems must specify the IPU explicitly because of the awareness of IPU.

Prepare write. In receiving an IPU, HORAE first allocates an available location from the preparatory area (p-area), a pre-reserved area of each device for handling dependency loops. It stores the location in the `plba` region (shown in Figure 6) of the ordering metadata. Next, it dispatches the IPU to the `plba` position of p-area, via the same routine as the classic orderless write. Further, HORAE uses an in-memory IPU radix tree to record the `plba` for following read operations to retrieve the latest data. The IPU tree accepts the logical block address (`lba`) as the input and outputs the newest `plba`. Compared to the IO processing, the modification on the IPU tree is performed in memory, and its overhead is thus negligible.

Commit write. HORAE applies the effects of IPUs via the commit write, once the durability of the prepare write is satisfied. HORAE firstly scans the ordering queue, and merges the overlapping data blocks of the p-area. Then, it writes the merged data blocks back to their original destination concurrently. When the commit write completes, HORAE removes associated entries of the IPU tree, and dequeues the entries between the `head` and `flushed` pointer of the ordering queue.

HORAE introduces two commit write policies, lazy and ea-

API	Explanation
olayer_init_stream(sid, param)	Register an ordered write stream with ID `sid` and parameters `param`
olayer_submit_bio(bio, sid)	Submit an ordered block IO `bio` to stream `sid`
olayer_submit_bh(bh, op, opflags, sid)	Submit a buffer head `bh` to stream `sid` with specific flags `opflags` and `op`
olayer_submit_bio_ipu(bio, sid)	Submit an ordered in-place update block IO `bio` to stream `sid`
olayer_blkdev_issue_flush(bdev, gfp_mask, error_sector, sid)	Issue a joint `FLUSH` to device `bdev` and stream `sid`
fbarrier(fd)	Write the data blocks and file system metadata of file `fd` in order
fdatabarrier(fd)	Write the data blocks of file `fd` in order
io_setup(nr_events, io_ctx, sids)	Create an asynchronous I/O context `io_ctx` and a set of streams `sids`
io_submit_order(io_ctx, nr, iocb, sid)	Write `nr` data blocks defined in `iocb` to stream `sid`

Table 1: The APIs of HORAE. HORAE *provides the* ***stream*** *(or sequencer) abstraction for upper layer systems. Each stream is identified by a unique* `sid`*, and represents a sequence of ordered IO requests. To realize multiple streams,* HORAE *evenly partitions the CMB area and p-area, and assigns a portion to each stream.*

ger commit write. The lazy commit write is performed in background when the IO stack is idle. When HORAE runs out of p-area space, it triggers eager commit write.

Read. As the ordered IPUs are redirected to p-area, the following read operation retrieves the latest data blocks from p-area first. HORAE searches the IPU tree for the `plba`, and reads the data from the `plba` position of p-area.

With write redirection for IPUs, HORAE provides higher consistency than the data consistency, which matches the default ordered mode of ext4. The data consistency requires that (1) the file system metadata does not point to garbage data, and (2) the old data blocks do not overwrite new ones. HORAE is able to preserve the order between data and file system metadata, and thus the file system metadata always points to valid data. Also, HORAE controls the order of parallel IPUs. Therefore, HORAE supports data consistency.

However, the data consistency does not provide request atomicity. Under data consistency, the IPUs directly overwrite valid data blocks, and can sometimes leave the file system a combination of old and new data, which brings inconvenience to maintain application-level consistency [2]. This is because commodity storage does not provide atomic write operations. For a write request containing a 4 KB data block, it may be split into multiple 512 B (determined partially by `Max_Payload_Size`) PCIe atomic ops (i.e., transactions) [17]. A crash may result in partially updated 4 KB data block.

HORAE enhances data consistency with request atomicity; continuous data blocks of each write request sent to the HORAE are made durably visible atomically due to double write. Once the durability of the prepare write is satisfied, HORAE ensures request atomicity. Otherwise, the prepare write may be partially completed and its effect is thus ignored.

5.5 Crash Consistency

In the face of a sudden crash, HORAE must be able to recover the system to a correct state that the data blocks are persisted in the correct order and the already durable data blocks with a completion response are not lost. This subsection discusses the crash consistency of HORAE, including the ordering queue consistency and the data block consistency.

The ordering queue. As stated in §5.2, HORAE writes the ordering metadata via MMIO. But MMIO writes are not guaranteed to be durable because they are posted transactions without completions. After each MMIO write, HORAE issues a non-posted MMIO read of zero byte length to ensure prior writes are made durable [24].

Upon a power outage, the ordering queue, the `head`, `flushed` and `tail` pointers, are saved to a backup region of flash memory, with the assistance of capacitors inside the SSD. When power resumes, HORAE loads the backup region into the CMB.

The data block. HORAE recovers the storage to a correct state with the support of the ordering queue. HORAE scans the ordering queue from the `head` position to the `flushed` position, in case of the data blocks are made durable with the `FLUSH` response but expired entries are not dequeued. HORAE commits the data blocks of the p-area that obey the order.

Further, HORAE recovers the durable yet not flushed data blocks (e.g., in SSD with PLP). Starting from the `flushed` position, HORAE sequentially scans the ordering queue, and commits the prepare writes until it finds a non-durable data block (i.e., `dr` bit is not set). After that, HORAE discards the following data blocks through filling the blocks with "zeros", because they violate the write order.

5.6 The API of HORAE and Use Cases

This subsection first describes the API of HORAE, and then presents two use cases: a file system leveraging a single write stream (i.e., a single ordering queue), and a user-space object store running with multiple write streams.

5.6.1 The API of HORAE

To enable the developers to leverage the efficient ordering control of HORAE, we provide three levels of functionalities/APIs shown in Table 1: the kernel block device, the file system and the asynchronous IO interface (i.e., libaio [3]).

Kernel block device interface. The kernel systems (e.g., file systems) can use `olayer_init_stream` to initialize an ordered write stream for further use. `olayer_submit_bio` and `olayer_submit_bh` deliver the same block IO sub-

mission function as classic interfaces (i.e., `submit_bio` and `submit_bh`); but they return when associated ordering metadata are persisted in the `sid` ordering queue (§5.2). `olayer_submit_bio_ipu` is for in-place updates (§5.4). `olayer_blkdev_issue_flush` is extended from classic `FLUSH` interface (i.e., `blkdev_issue_flush`); it keeps the same arguments and performs joint `FLUSH` on a given stream and target devices (§5.3).

File system interface. We offer two ordering file system interfaces, the `fbarrier` and `fdatabarrier`, for upper layer systems or applications to organize their ordered data flow. The `fbarrier` bears the same semantics as the `osync` of OptFS and `fbarrier` of BarrierFS; it writes the data blocks and journaled metadata blocks in order but returns without ensuring durability. The `fdatabarrier` is the same as that of BarrierFS; it only ensures the ordering of the application data blocks but not the journaled blocks. We further show the internals of these interfaces in the following §5.6.2.

Libaio interface. Libaio provides wrapper functions for async IO system calls, which are used for some systems (e.g., BlueStore and KVell [36]) designed on high-performance drives. We expose the ordering control path of HORAE via two new interfaces on libaio. `io_setup` allows developers to allocate a set of streams defined in `sids`. `io_submit_order` performs ordered IO submission; it bears the same semantics of `fdatabarrier`. We further show the usage of these interfaces in boosting BlueStore in §5.6.3.

Porting guidelines. We provide three guidelines. First, upper layer systems can distinguish the ordered writes from the orderless ones based on the categories of the request, and send them using HORAE's APIs. The requests that contain metadata, the write-ahead log and the data of eager persistence (e.g., data specified by the `fsync()` thread) are treated as ordered writes. Second, due to the separation of ordering control path, upper layer system can design and implement the ordering and durability logic individually. In the ordering logic, they can use the ordering control interface (e.g., `olayer_submit_bio`) to dispatch the following ordered writes immediately after the previous one returns from the control path. This allows the ordered data blocks to be transferred in an asynchronous manner without waiting for the completion of DMA. Third, they can remove all `FLUSH`es that serve as ordering points in the ordering logic, and invoke an eventual joint `FLUSH` in the durability logic to guarantee durability.

5.6.2 The HORAEFS File System

We build HORAEFS atop the HORAE with a set of modifications of BarrierFS [46]. BarrierFS builds on Ext4, and divides the journaling thread (i.e., JBD2 in Figure 8) into submit thread and flush thread.

HORAEFS inherits this design, and the major changes are that (1) we submit ordered writes and reads to the ordering layer first, (2) we remove the `FLUSH` to coordinate the data and journal device and (3) we detect IPUs through inspecting

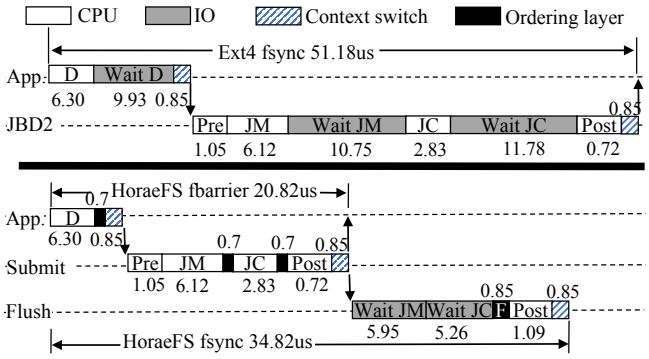


Figure 8: The `fsync()` and `fbarrier()`. *4 KB data size. The number shows the latency of each operation in microseconds. D: application data blocks. Pre: prepare the journaled metadata. JM: journaled file system metadata. JC: journaled commit record. Post: change transaction state, calculate journal stats, etc.*

the `BH_New` states of each write. Although the journal area is repeatedly overwritten, we do not treat the journaled writes as IPUs, as the journal area is always cleaned before reused.

Figure 8 shows a side-by-side comparison between Ext4 and HORAE on a NVMe SSD. For each 4 KB allocating write in most cases, both file systems issue three data blocks, namely the data block (*D*), the journaled metadata block (*JM*) and the journaled commit block (*JC*). Further, both file systems order the data blocks with $\{D, JM\} \leq JC$. Ext4 enforces the ordering constraints through the exclusive IO processing. It waits for the completion of preceding data blocks (e.g., Wait JM) and issues a `FLUSH` (not shown in the figure because it is returned by the block layer quickly). HORAEFS eliminates the exclusive IO processing, and preserves the order through the ordering layer, as shown in the black rectangle. HORAEFS waits for the durability of the associated blocks in flush thread, and finally issues a `FLUSH` to the ordering layer for durability.

HORAEFS differs from BarrierFS in the IO dispatching (i.e., the white rectangle) and IO waiting (i.e., the gray rectangle) phase. During IO dispatching phase, BarrierFS passes through the entire order-preserving SCSI software stack. Besides, BarrierFS experiences extra waiting time due to the order-preserving hardware write, i.e., the barrier write.

5.6.3 The HORAESTORE Distributed Storage Backend

We build HORAESTORE atop the HORAE with a set of modifications of BlueStore [23], an object store engine of Ceph.

BlueStore directly manages the block device by async IO interfaces (e.g., libaio), providing transaction interfaces for distributed object storage (i.e., RADOS). Inside each write transaction, it first persists the aligned write to data storage, followed by storing the unaligned small writes and metadata in a RocksDB [1] KV transaction (KVTXN). The RocksDB first writes the write-ahead log (WAL), then applies the updates to the KV pairs. For inter-transaction ordering, it uses sequencers; the next transaction can not start a KVTXN until the preceding one has made the aligned write durable and

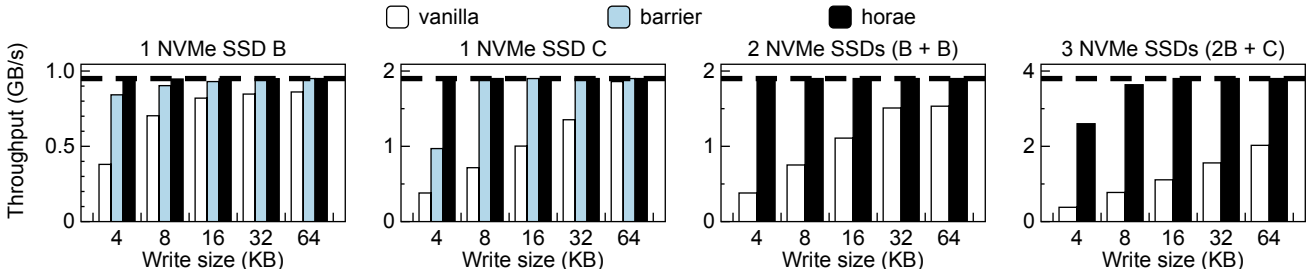


Figure 9: Ordered Write Performance. *The throughput of randomly writing 1 GB drive space by 1 write stream (i.e., 1 thread). vanilla: native Linux NVMe IO stack. horae:* HORAE *IO stack. barrier: Barrier-enabled IO stack. Horizontal dotted lines: maximum device bandwidth.*

started its KVTXN. Besides, BlueStore uses a single KV thread to serially perform KVTXNs, which blocks and waits for in-progress KVTXNs to become durable.

HORAESTORE accelerates the transaction ordering guarantee, while exporting the same transaction interfaces. For intra-transaction ordering, HORAESTORE uses the new async IO interface of HORAE, `io_submit_order()`, to control the write order of data, WAL and KV pairs. With this new interface, for each transaction, HORAESTORE processes the data, WAL and KV pairs concurrently.

For inter-transaction ordering, HORAESTORE extends the overlapping range of dependent transactions, and improves the concurrency of KVTXNs' submission. First, in HORAESTORE, the following transaction starts the KVTXN immediately after the preceding one has satisfied the ordering of its aligned writes (D) and KVTXN. In other words, HORAE can process two dependent transactions concurrently, once the ordering between them is satisfied, i.e., $\{D_1, D_2\} \preceq KVTXN_1 \preceq KVTXN_2$. Second, HORAESTORE separates the ordering of KVTXN from durability; it starts the KVTXNs in a KV submit thread for ordering, and ensures the durability in a KV flush thread. Hence, more KVTXNs can be queued in the KV submit thread, and can further be dispatched to RocksDB for processing.

5.7 Implementation Details and Discussion

We implement HORAE in Linux kernel as a pluggable kernel module (i.e., the ordering layer), consisting of 1288 lines of code (LOC); no changes are needed for traditional IO stack (i.e., the block layer, NVMe and SCSI driver). HORAEFS is implemented based on BarrierFS with approximately 100 LOC changes. HORAESTORE is implemented based on BlueStore with around 200 LOC change.

HORAE needs a region of byte-addressable persistent memory for efficient ordering control path. We realize this by remapping the CMB region of a capacitor-backed CMB-enabled SSD from StarBlaze [20] using `ioremap_wc()`.

Currently, CMB-enabled SSDs are already available in the market [14, 18]. Moreover, many SSDs have enabled power loss protection [10, 12, 18, 32]. Therefore, the persistent CMB (or PMR) requirement of HORAE can be achieved easily. We further discuss the alternatives of the CMB in §7.

6 Evaluation

This section evaluates the HORAE, HORAEFS and HORAESTORE by answering the following questions:

- What is the performance of HORAE in guaranteeing the ordering? (§6.2, §5.2)
- What is the performance of HORAE in guaranteeing the durability? (§6.3, §5.3)
- How does HORAE perform under in-place updates with different consistency level? (§6.4, §5.4)
- Can HORAE recover correctly after a crash and how much overhead does recovery introduce? (§6.5, §5.5)
- How much improvement does HORAE bring to applications? (§6.6, §5.6)

6.1 Experimental Setup

Hardware. We conduct all experiments with a 12-core machine running at 2.50 GHZ. Table 2 shows the specification of the candidate SSDs. We use three broadly categorized SSDs: the SATA SSD (labelled as A), the consumer-grade NVMe SSD (B), and the high-performance datacenter-grade NVMe SSD (C). The NVMe SSD B and C are with PLP. The size of CMB used by HORAE is 2 MB.

Compared Systems. We mainly compare with two types of IO stacks, Vanilla and BarrierIO [46]. Vanilla is the default Linux IO stack. Ext4 [7] is a journaling file system running upon vanilla. We setup ext4 with default options in ordered journaling mode (denoted as ext4-DR). We disable the barriers in ext4-DR (`nobarrier` option) with ext4-OD, which only guarantees the dispatch order reaching in storage buffer (not storage medium). Similar to ext4, we test BarrierFS [46] upon BarrierIO stack with durability guarantee (denoted as BFS-DR) and ordering guarantee (denoted as BFS-OD). Since we do not have barrier compliant storage

	Model	Seq. Bandwidth	Rand. IOPS (8GB span)
A	Samsung 860 PRO SATA	Read: 560MB/s Write: 530MB/s	Read: 100k Write: 90k
B	Intel 750 NVMe	Read: 2200MB/s Write: 950MB/s	Read: 430K Write: 230K
C	Intel DC P3700 NVMe	Read: 2800MB/s Write: 1900MB/s	Read: 640K Write: 475K

Table 2: SSD Specifications.

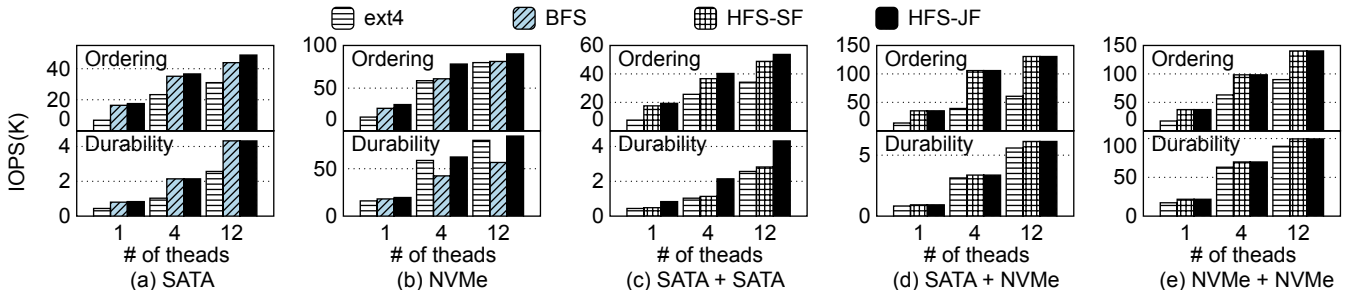


Figure 10: File System Performance under FIO Allocating Write Workload. *Ordering:* `nobarrier` *option of ext4, but only guarantees a relaxed ordering of reaching the storage write buffer.* `fbarrier()` *of BarrierFS and* HoraeFS. *Durability:* `fsync()`. *SF: serialized* `FLUSH`. *JF: joint* `FLUSH`.

devices, we reuse the software of BarrierFS and add extra 5% overhead of the hardware barrier write, following the assumption of the BarrierFS paper. To run BarrierFS correctly in multi-queue drives, we modify the NVMe driver to setup only one IO command queue. HoraeFS (abbreviated as HFS) uses just one ordering queue.

6.2 Basic Performance Evaluation

First, we demonstrate the effectiveness of Horae in guarantee the ordering (§5.2), through measuring the throughput of three IO stacks on block devices. We vary the number of devices and organize them as soft-RAID 0. Specifically, we evenly distribute X KB random writes to different devices in a round-robin fashion. Figure 9 shows the overall throughput of ordering guarantee with varying the write size. Note that we only have the result of BarrierIO on a single device, because it does not support multiple drives.

Result. From the result, we find that Horae outperforms vanilla and BarrerIO IO stack by 4.1× and 2× respectively in the case of a single device and 4 KB write unit. On multiple devices, Horae achieves up to 6.8× throughput than vanilla. Further, we observe that Horae can easily saturate the device bandwidth with small write units (e.g., 4 KB).

Analysis. We now decompose the IO path to better understand the performance. The overall IO path can be broken down into four parts, and we measure the overhead of each part when issuing 4 KB data blocks as follows: (1) data page preparation costs 0.8 us in our test; (2) the ordering layer processes and writes ordering metadata within 0.7 us; (3) about 1.0 us is spent on block layer, which performs request merging and `bio`(block IO data structure)-to-`request`(NVMe driver data structure) transmission; (4) data DMA, device-side processing and interrupt handler consume 8.7 us. The classic approach experiences all parts except (2), which counts up to 10.5 us in total. Thus, the maximum IOPS and throughput that a single write stream can achieve in classic IO stack are 95K and 380 MB/s. Horae experiences (1) and (2) in most cases. The ordering layer delegates the submission of orderless block IO to per-CPU background submitter threads, so as to hide the overhead of block layer for foreground ordering calls. Thus, Horae can achieve up to 2.6 GB/s in the case of 4 KB writes. BarrierIO eliminates (4) in ordering guarantee, and thus can achieve up to 2.2 GB/s in NVMe stack theoretically. However, we observe that configuring the drive to a single IO command queue considerably decreases the available bandwidth of high-performance storage.

6.3 File System Evaluation

In this subsection, we evaluate the performance of POSIX file systems atop different IO stacks with varying the number and type of storage devices. Moreover, we demonstrate the effectiveness of Horae in guaranteeing the durability (§5.3). We perform allocating write so that the file system always finds the updated metadata in the journal. We set up the file system with two modes: the internal journal that mixes data and journal blocks in a single device, and the external journal that uses a dedicated device for locating journal. The result is shown in Figure 10, and we make the following observations.

Effect of removing flush. As shown in Figure 10(a), on SATA SSD, HFS achieves 80% higher IOPS averagely against ext4, and exhibits comparable performance compared to BFS. Here, the major overhead is two `FLUSH`es. The former one is used to control the write order of the data blocks and the commit record, and the later one is to ensure durability. The `FLUSH` of SATA SSD exposes raw flash programming delay (1 ms). In BFS and HFS, the former `FLUSH` is eliminated.

Effect of async DMA. As shown in Figure 10(b), on NVMe SSD, HFS achieves 22% higher IOPS than ext4. Compared to the durability of HFS, the ordering guarantee of HFS can further boost IOPS by 57%. The major overhead here is shifted to the DMA transfer. Figure 8 shows the source of improvement, and the number next to each rectangle shows the single thread latency of each operation. Due to the separation of the ordering and durability, HFS and BFS can overlap the DMA transfer with CPU processing, and thus partly hide the DMA delay. BFS does not perform well on NVMe SSD due to the restriction of the single IO command queue, especially when we increase the number of threads.

Effect of joint flush. As shown in Figure 10(c), HFS outperforms ext4 by 88% and 90% on average in durability and ordering respectively. Comparing HFS-PF to HFS-SF, we find the joint flush improves the overall performance by up to

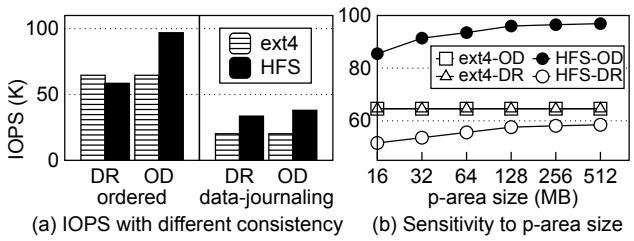


Figure 11: In-Place Update Performance. *OD: `nobarrier` option of ext4, `fbarrier()` of HFS. DR: `fsync()`. Test on SSD C.*

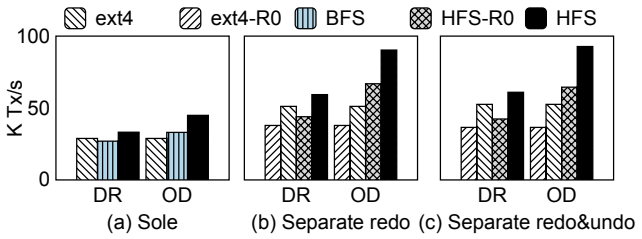


Figure 12: MySQL under OLTP-insert Workload. *(a) Sole: mix data, redo log and undo log in device B. (b) Separate redo: data and undo log to device B, redo log to device C. (c) Separate redo & undo: data to device B, redo log to device C, undo log to another device B. DR: The `fsync()` used to control the write order of transactions is replaced with `fbarrier()`. OD: All `fsync()`s are replaced with `fbarrier()`s. `Sync()` the database every 1 second. R0: organize underlying devices as logical volumes using RAID 0.*

70%. This is because joint flush allows physically independent devices to perform flushing concurrently.

Effect of parallel device access. Figure 10(d-e) plots the result when we use an NVMe SSD to accelerate the journal. In ordering, HFS improves ext4 by 150% and 76% respectively. The major contributor here is the parallel device access. In HFS, once the associated ordering metadata is processed serially by the control path, the data blocks can be transferred and processed by individual devices concurrently. While in ext4, this is done in a serialized manner.

6.4 In-Place Update Evaluation

In this subsection, we evaluate the performance of in-place update under different consistency and with varying the size of the preparatory area (p-area) (§5.4).

As shown in the X title of Figure 11(a), we first setup the file system with two modes, the ordered and data-journaling mode, representing data and version consistency, respectively. Then, we issue 10 GB overwrites to a 10 GB file. The ordered mode performs metadata journal. The data-journaling mode performs data journal to achieve the version consistency that the version of data matches that of metadata.

In the ordered mode, the double write of eager commit write enhances the data consistency at the cost of 10% IOPS loss in durability. In ordering, HFS exhibits 50% higher IOPS compared to ext4, because HFS can emit multiple IPUs simultaneously without interleaving each IPU with DMA transfer and `FLUSH` command.

In the data-journaling mode, both file systems first put the IPU to journal area. When performing journal, HFS dispatches the commit record immediately after the journaled data, and thus provides 60% higher IOPS on average.

When HORAE runs out of p-area space, HORAE blocks incoming requests and triggers eager commit write. To investigate the performance of HORAE in such a situation, we run the same IPU workload with the scaling of p-area size. The results are shown in Figure 11(b). We find that HFS-OD performs dramatically better than ext4-OD even with small p-area. To provide request atomicity, HFS-DR delivers less IOPS than ext4-DR. As we enlarge the p-area, the IOPS gap narrows.

6.5 Crash Recovery Evaluation

To verify the consistency guarantees of HORAE (§5.5), we run workloads, and forcibly shut down the machine at random. We restart the machine and measure the recovery performance. We choose Varmail workload of Filebench [8] for its intensive `fsync()`. Varmail contains two `fsync()`s in each flow loop, and we replace the first one with `fbarrier()`.

We repeat the test 30 times, and observe HORAEFS can always recover to a consistent state. The recovery time comes from two main parts: the ordering queue load time and commit write time. First, HORAE loads the pointers and the ordering metadata into host DRAM, which consumes 29.8 ms on average. Next, the commit write requires "read-merge-write", and costs 497.6 ms on average.

6.6 Application Evaluation

6.6.1 MySQL

We evaluate MySQL with OLTP-insert workload [21]. The setups are described in the caption of Figure 12.

In sole configuration, HFS-DR outperforms ext4-DR and BFS-DR by 15% and 23% respectively. In ordering, HFS-OD prevails ext4-OD by 56% and achieves 36% higher TPS than BFS-OD. This evidences that HFS is more efficient in controlling the write order.

When using dedicated devices to store redo and undo logs (i.e., Figure 12(b)), HFS-DR outperforms ext4-DR by 16%, and HFS-OD performs 76% better than ext4-OD. This is because HFS can parallelize the IOs to individual devices.

Comparing Figure 12(b) with (c), we find that separating undo logs does not bring much improvement in both ext4 and HFS. Undo logs perform logical logging to retain the old version of database tables, which incurs less writes compared to physical logging (redo log). MySQL tightly embeds the undo logs in the table files, thus separating undo logs does not alleviate the write traffic to the data device.

Comparing HFS with HFS-R0, we observe that, from the performance aspect, manually distributing data flows to de-

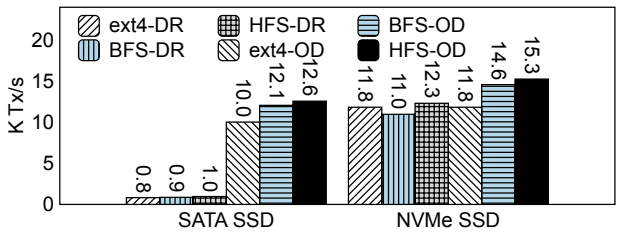


Figure 13: SQLite Random Insert Performance. *SQLite runs at WAL mode. 1M inserts in random key order. Key size 16 bytes, value size 100 bytes. DR: The first three* `fdatasync()`*s used to control storage order of transactions are replaced with* `fdatabarrier()`*s, but the last one remains intact. OD: All* `fdatasync()`*s are replaced with* `fdatabarrier()`*s.*

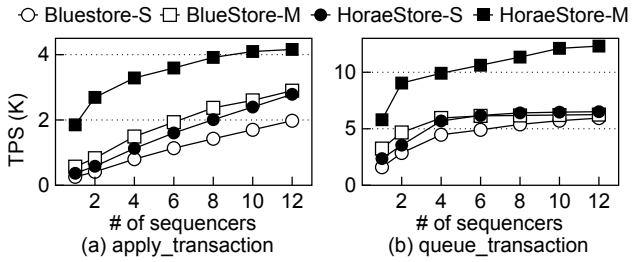


Figure 14: Object Store Performance. *Store-S: mix data, metadata, WAL to device A. Store-M: data to device A, metadata to device B, WAL to device C. apply_transaction: durability guarantees. queue_transaction: ordering guarantee.*

vices of particular usage is better than the automatic dispersion of logical volumes. A naive implementation of RAID 0 treats the devices equally. However, the data flows of application usually have different write traffic and locality. Therefore, uniform distribution potentially bounds the better devices and ruins the data locality.

6.6.2 SQLite

This subsection focuses on the performance of SQLite [19]. The detailed setups are presented in the caption of Figure 13.

On SATA SSD, the ordering setups (OD) outperform the durability ones (DR) by an order of magnitude due to the reduction of the prohibitive `FLUSH`. In ordering, both BFS and HFS exhibit over 20% performance gain against ext4 due to the separation of ordering and durability that brings chances of overlapping CPU with IOs.

On NVMe SSD, as the `FLUSH` becomes inexpensive, ext4-OD achieves almost the same performance as ext4-DR. HFS separates the control path from the data path, and thus SQLite can order the table files and logs through `fbarrier()`. Therefore, more IOs can be processed at the same time.

6.6.3 BlueStore

This subsection evaluates the transaction processing of object store with default options. We use the built-in object store benchmark [5] of Ceph with varying the number of

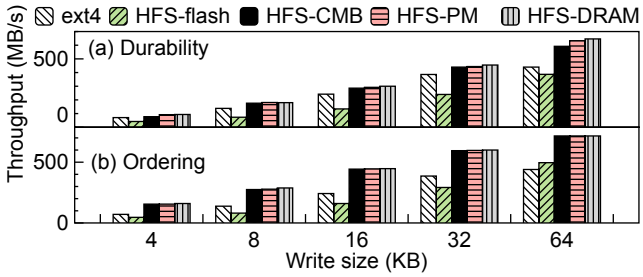


Figure 15: Comparison of The Media of The Ordering Queue. *flash: block IO to SSD. CMB: MMIO to SSD's memory buffer. PM: Intel Optane persistent memory. DRAM: capacitor-back memory.*

sequencers. Each sequencer serializes the transactions, and transactions of different sequencers do not have dependency. Each transaction issues 20 KB write, which is split into 16 KB aligned write to data device and 4 KB small write to RocksDB. Two interfaces are evaluated, `apply_transaction()` and `queue_transaction()`, representing ordering and durability guarantee, respectively. Figure 14 shows the results.

In Figure 14(a), HoraeStore exhibits 1.4× and 2.1× TPS gain against BlueStore, in S and M setup, respectively. To preserve order, BlueStore does not submit the small write and metadata to RocksDB until the aligned write has been completed. While in HoraeStore, once the aligned write and KV transaction have been processed serially via the control path, associated data blocks can be processed concurrently.

The `queue_transaction()` brings opportunities to apply multiple transactions. As shown in Figure 14(b), HoraeStore outperforms BlueStore averagely by 23% and 83% in S and M setup, respectively. Due to the write dependency over multiple devices, the slower data device burdens the faster metadata and WAL devices. Hence, BlueStore-M delivers similar TPS compared to BlueStore-S. In HoraeStore-M, as the control path guarantees the ordering, the synchronization between aligned write and KV transaction is alleviated. Further, HoraeStore enables more KV transactions to continuously fulfill the metadata and WAL storage.

7 Discussion

CMB Alternatives. Recall that Horae persists the ordering metadata in the CMB for efficiency. Nevertheless, several off-the-shelf non-volatile media are capable of storing the ordering metadata: SSD (flash), Intel persistent memory (PM) and capacitor-backed DRAM. We locate the ordering queue in these media and measure the single-threaded throughput of the ordered writes, as shown in Figure 15. We find that storing the ordering metadata directly through the block-based interface to SSD (i.e., HFS-flash) significantly decreases the throughput. This is because, even the ordering metadata is 16 B, it must be padded to 4 KB, where the 4 KB synchronous PCIe transfer masks the concurrency of translated orderless writes. When the write size increases (over 64 KB), HFS-flash gradually outperforms ext4. We also find the PM and

DRAM are also satisfiable alternatives of the CMB.

8 Related Work

Storage order. Many researchers [25–28, 40, 46] have studied and mitigated the overhead of storage order guarantee. Among these studies, the closest ones are OptFS [26] and BarrierFS [46] that separate the ordering guarantee from durability guarantee and provide similar ordering primitive. OptFS, BarrierFS and HoraeFS are proposed under different storage technologies (HDD, SATA SSD, NVMe SSD and storage arrays), thereby mainly differing in the architecture, the scope of application and hardware requirements. First, OptFS embeds the transaction checksum in the journal commit block and detects the ordering violation during recovery, which reduces the rotational disk `FLUSH` overhead at runtime. BarrierFS preserves the order layer by layer, thereby aligning with the single queue structure of SCSI stack and devices. They are difficult to be extended to multiple queues or multiple devices. In contrast, Horae stores the ordering metadata via a dedicated control path to maintain the write order. This design aims to let the ordering bypass the traditional stack to enable high throughput and easy scaling to multiple devices. Second, the checksum-based ordering approach of OptFS is limited to continuous address space (e.g., file system journaling), because the checksum can be only used to detect the ordering violation of data blocks in pre-determined locations. Alternatively, Horae builds a more generic ordering layer which can spread data blocks to arbitrary logical locations of any device. Third, OptFS requires the disk to support asynchronous durability notification. BarrierIO requires barrier compliant storage device which is only available in a few eMMC (embedded multimedia card) products. Horae can run on the standard NVMe devices with exposed CMB. The CMB feature is already defined in NVMe spec, and is under increasing promotion and recognition by NVMe and SPDK communities [6].

Dependency tracking. Some works use dependency tracking techniques to handle storage order. Soft updates [39] directly tracks the dependencies of the file system structures in a per-pointer basis. Similarly, Featherstitch [28] introduces the *patch* to specify how a range of bytes should be changed. Horae also tracks the write dependencies in the ordering queue. The tracking unit of Horae is different from prior works; each entry in Horae describes how a range of data blocks (e.g., 4 KB) should be ordered. The block-aligned tracking introduces less complexity of both dependency tracking and file system modifications, and it is more generic in the context of block device. In addition, due to the disability of telling data versions, soft updates does not support version consistency. Featherstitch assumes single in-progress write to the same block address, and treats dependency loop as errors. Thus, the in-place updates of Featherstitch must wait for the completion of preceding one. Instead, Horae saves the in-place updates from long DMA transfer through write redirection with enhanced consistency.

Storage IO stack. A school of works [30,31,34,35,37,40–43, 49,51] improve the storage IO stack. Xsyncfs [40] uses output-triggered commits to persist a data block only when the result needs to be externally visible. IceFS [37] allocates separate journals for each container for isolation. SpanFS [31] partitions the file system at *domain* granularity for performance scalability. Built atop F2FS [34], ParaFS [51] co-designs the file system with the SSD's FTL to bridge the semantics gap between the hardware and software. iJournaling [41] designs fine-grained journaling for each file, and thus mitigates the interference between `fsync()` threads. CCFS [42] provides similar *stream* abstraction at file system level for applications to implement correct crash consistency. Its stream is designed on individual journals and still relies on exclusive IO processing to preserve the order. Horae exports stream at block level via the dedicated control path, not relying on exclusive IO processing nor journal. TxFS [30] leverages the file system journaling to provide transactional interface. Son *et al.* [43] propose a high-performance and parallel journal scheme. FlashShare [49] punches through the IO stack to device firmware to optimize the latency for ultra-low latency SSDs. AsyncIO [35] overlaps the CPU execution with IO processing, so as to reduce the `fsync()` latency. CoinPurse leverages the byte interface and device-assisted logic to expedite non-aligned writes [47]. However, these works still rely on exclusive IO processing to control the internal order (e.g., the order between data blocks and metadata blocks) and external order (e.g., the order of applications' data).

NoFS [27] introduces backpointer-based consistency to remove the ordering point between two dependent data blocks. Due to the lack of ordering updates, NoFS does not support atomic operations (e.g., `rename()`).

9 Conclusion

In this paper, we revisit the write dependency issue on high-performance storage and storage arrays. Through a performance study, we notice that with the growth of performance of storage arrays, the performance loss induced by the write dependency becomes more severe. Classic IO stack is not efficient in resolving this issue. We thus propose a new IO stack called Horae. Horae separates the ordering control from the data flow, and uses a range of techniques to ensure both high performance and strong consistency. Evaluations show that Horae outperforms existing IO stacks.

10 Acknowledgement

We sincerely thank our shepherd Vijay Chidambaram and the anonymous reviewers for their valuable feedback. We also thank Qing Wang and Zhe Yang for the discussion on this work. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61832011, 61772300).

References

[1] A Persistent Key-Value Store for Fast Storage. https://rocksdb.org/.

[2] A way to do atomic writes. https://lwn.net/Articles/789600/.

[3] An async IO implementation for Linux. https://elixir.bootlin.com/linux/v4.18.20/source/fs/aio.c.

[4] BFQ (Budget Fair Queueing). https://www.kernel.org/doc/html/latest/block/bfq-iosched.html.

[5] Ceph Objectstore benchmark. https://github.com/ceph/ceph/blob/master/src/test/objectstore_bench.cc.

[6] Enabling the NVMe™ CMB and PMR Ecosystem. https://nvmexpress.org/wp-content/uploads/Session-2-Enabling-the-NVMe-CMB-and-PMR-Ecosystem-Eideticom-and-Mell....pdf.

[7] ext4 Data Structures and Algorithms. https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html.

[8] Filebench - A Model Based File System Workload Generator. https://github.com/filebench/filebench.

[9] fio - Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[10] Intel Solid State Drive 750 Series Datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-750-spec.pdf.

[11] Intel® SSD 545s Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/5-series/ssd-545s-series/545s-256gb-2-5inch-6gbps-3d2.html.

[12] Intel® SSD DC P3700 Series. https://ark.intel.com/content/www/us/en/ark/products/79621/intel-ssd-dc-p3700-series-2-0tb-2-5in-pcie-3-0-20nm-mlc.html.

[13] MySQL reference manual. https://dev.mysql.com/doc/refman/8.0/en/.

[14] NoLoad U.2 Computational Storage Processor. https://www.eideticom.com/uploads/attachments/2019/07/31/noload_csp_u2_product_brief.pdf.

[15] NVMe specifications. https://nvmexpress.org/resources/specifications/.

[16] NVMe SSD with Persistent Memory Region. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170810_FM31_Chanda.pdf.

[17] PCI Express Base Specification Revision 3.0. http://www.lttconn.com/res/lttconn/pdres/201402/20140218105502619.pdf.

[18] Product Brief: Intel® Optane™ SSD DC D4800X Series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-dc-d4800x-series-brief.html.

[19] SQLite. https://www.sqlite.org/index.html.

[20] Starblaze OC SSD. http://www.starblaze-tech.com/en/lists/content/id/137.html.

[21] SysBench manual. https://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[22] Embedded Multimedia Card. http://www.konkurel.ru/delson/pdf/D93C16GM525(3).pdf, 2018.

[23] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[24] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong. 2b-ssd: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 425–438. IEEE Press, 2018.

[25] Y.-S. Chang and R.-S. Liu. Optr: Order-preserving translation and recovery design for ssds with a standard block device interface. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 1009–1023, Berkeley, CA, USA, 2019. USENIX Association.

[26] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[27] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 9, USA, 2012. USENIX Association.

[28] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized

file system dependencies. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 307–320, New York, NY, USA, 2007. Association for Computing Machinery.

[29] M. Hedayati, K. Shen, M. L. Scott, and M. Marty. Multi-queue fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, Renton, WA, July 2019. USENIX Association.

[30] Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel. Txfs: Leveraging file-system crash consistency to provide ACID transactions. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 879–891, Boston, MA, July 2018. USENIX Association.

[31] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. Spanfs: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 249–261, Berkeley, CA, USA, 2015. USENIX Association.

[32] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh. Durable write cache in flash memory ssd for relational and nosql databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 529–540, New York, NY, USA, 2014. ACM.

[33] N. Kirsch. Phison E12 High-Performance SSD Controller. https://www.legitreviews.com/sneak-peek-phison-e12-high-performance-ssd-controller_206361, 2018.

[34] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.

[35] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong. Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, Renton, WA, July 2019. USENIX Association.

[36] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[37] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 81–96, Berkeley, CA, USA, 2014. USENIX Association.

[38] Marvell. Marvell 88SS1093 Flash Memory Controller. https://www.marvell.com/content/dam/marvell/en/public-collateral/storage/marvell-storage-88ss1093-product-brief-2017-03.pdf, 2017.

[39] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, page 24, USA, 1999. USENIX Association.

[40] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 1–14, USA, 2006. USENIX Association.

[41] D. Park and D. Shin. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 787–798, Berkeley, CA, USA, 2017. USENIX Association.

[42] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 181–196, Santa Clara, CA, Feb. 2017. USENIX Association.

[43] Y. Son, S. Kim, H. Y. Yeom, and H. Han. High-performance transaction processing in journaling file systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 227–240, Berkeley, CA, USA, 2018. USENIX Association.

[44] S. C. Tweedie. Journaling the linux ext2fs filesystem. In *In LinuxExpo'98: Proceedings of The 4th Annual Linux Expo*, 1998.

[45] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. pages 307–320, 11 2006.

[46] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho. Barrier-enabled io stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 211–226, Berkeley, CA, USA, 2018. USENIX Association.

[47] Z. Yang, Y. Lu, E. Xu, and J. Shu. Coinpurse: A device-assisted file system with dual interfaces. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[48] J. Yeon, M. Jeong, S. Lee, and E. Lee. Rflush: Rethink the flush. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 201–209, Berkeley, CA, USA, 2018. USENIX Association.

[49] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, and M. Jung. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, Carlsbad, CA, Oct. 2018. USENIX Association.

[50] J. Zhang, M. Kwon, M. Swift, and M. Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 121–136, Santa Clara, CA, Feb. 2020. USENIX Association.

[51] J. Zhang, J. Shu, and Y. Lu. Parafs: A log-structured file system to exploit the internal parallelism of flash devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, Denver, CO, June 2016. USENIX Association.

Blockene: A High-throughput Blockchain Over Mobile Devices

Sambhav Satija[⋆], Apurv Mehra[⋆], Sudheesh Singanamalla[†*], Karan Grover[⋆],
Muthian Sivathanu[⋆], Nishanth Chandran[⋆], Divya Gupta[⋆], Satya Lokam[⋆]
[⋆]*Microsoft Research India* [†]*University of Washington*

Abstract

We introduce Blockene, a blockchain that reduces resource usage at member nodes by orders of magnitude, requiring only a smartphone to participate in block validation and consensus. Despite being lightweight, Blockene provides a high throughput of transactions and scales to a large number of participants. Blockene consumes negligible battery and data in smartphones, enabling millions of users to participate in the blockchain without incentives, to secure transactions with their collective honesty. Blockene achieves these properties with a novel split-trust design based on delegating storage and gossip to untrusted nodes.

We show, with a prototype implementation, that Blockene provides throughput of 1045 transactions/sec, and runs with very low resource usage on smartphones, pointing to a new paradigm for building secure, decentralized applications.

1 Introduction

Blockchains provide a powerful systems abstraction: they allow mutually untrusted entities (*members*) to collectively manage a *ledger* of transactions in a decentralized manner.

All blockchains today require member nodes to run powerful servers with significant network, storage, and compute resources. Blockchains based on *proof-of-work* [5, 30] push resource usage to an extreme, requiring significant compute for puzzle-solving, but even consortium blockchains [13] and blockchains based on *proof-of-stake* [21] incur significant network and storage costs to keep the blockchain up to date at a high transaction throughput. Blockchains today are therefore limited to use-cases where members have a strong incentive to participate, and can hence afford the high resource cost. For example, in consortium blockchains [13], business efficiency improves, while in cryptocurrencies [21, 30], members earn currency.

Interestingly, the high resource requirement of blockchains also weakens *reliability* for several real-world applications. Blockchains require that majority (typically two-thirds) of members are honest, a property that is easier to guarantee when a large number of members participate. However, wide-scale adoption of a blockchain is hard given the high resource requirement, especially in scenarios where members do not have a direct incentive to participate. Not surprisingly, public blockchains with high membership today target cryptocurrencies [5, 30].

In this paper, we present *Blockene*[1], an ultra-lightweight, large scale blockchain that provides high throughput for real-world transactions. By being lightweight and scalable, it enables wide-scale adoption by millions of users. By enabling large scale of participation, *Blockene* makes it plausible to assume honest-majority. By being high-throughput, *Blockene* supports real-world transaction rates.

The key breakthrough in *Blockene* is that instead of requiring members to run powerful servers, *Blockene* is the first blockchain that enables members to participate as first-class citizens in consensus even while running on devices as lightweight as smartphones, lowering cost by orders of magnitude.

Network: Blockchains rely on peer-to-peer gossip between members; at a high transaction rate, gossip would require tens of GBs of data transfer per day; *Blockene* requires only about 60MB of data transfer per day on a smartphone.

Storage: Member nodes in blockchains keep a copy of the entire blockchain (terabytes at high-throughput); in *Blockene*, members incur only a few hundred MBs of storage.

Compute: Even the gossip cost of typical blockchains would drain battery on mobile nodes; *Blockene* ensures that battery drain is less than 3% per day.

Thus, a user incurs no perceptible cost while running *Blockene*. As the low resource usage in *Blockene* makes it feasible even in a smartphone, *Blockene* can also run on desktops, with much lighter resource usage than state-of-the-art.

Blockene achieves three conflicting properties: large scale of participation, high throughput, and lightweight resource usage, catering to even scenarios where there is no direct incentive (*e.g.*, altruistic participation), and handling transactions across variety of use-cases including those on public funds. A comparison of *Blockene* with other blockchain architectures is depicted in Table 1.

Example application: Audited Philanthropy. Charitable donations to non-profits are in excess of USD 500 billion *annually* worldwide [7, 8, 10]. However, from a donor's perspective, the lack of transparency on the end-use of funds makes donations vulnerable to sub-optimal use or mismanagement by non-profits, especially in regions where regulatory enforcement is ineffective or crippled by corruption. A sys-

*Sudheesh was with Microsoft Research India while doing this work.

[1]Named after *Graphene*, one of the lightest and strongest materials.

tem that provides a public, end-to-end trail of funds from the donor to the end beneficiary, will exert market pressure on non-profits, besides motivating donors. A blockchain can provide such tracking, but given the scale of funds involved, a small consortium of members cannot be trusted with operation of the blockchain. Ideally, such a blockchain should be jointly controlled by millions of citizens altruistically. Similar requirements arise in government/public spending.

Key techniques in Blockene: *Blockene* adopts a novel system design based on a split-trust architecture with a new security model. There are two types of nodes in *Blockene*: *Citizens* and *Politicians*. *Citizens* run on smartphones and are the real members of the blockchain, i.e., they have voting power in consensus protocol; hence we assume that two-thirds of the *Citizens* are honest (a reasonable assumption with millions of *Citizens*). On the other hand, *Politicians* run on servers and are untrusted, i.e., do not participate in consensus. *Politicians* are fewer in number (few hundreds), and we require *only 20%* of them to be honest. Although *Politicians* do the heavy work such as storing the blockchain, our protocols ensure that *Citizens* can detect and handle malicious behavior even if 80% of the *Politicians* collude with the one-third of malicious *Citizens*. *Citizens* deal with high dishonesty of *Politicians* by using a new primitive called *replicated verifiable reads*: the *Citizen* reads the same data from multiple *Politicians* and can get the correct value even if one (out of, say, 25) is honest.

Citizens perform transaction validation, and decide on the block and resulting global state to commit, by running Byzantine consensus. To make consensus feasible with millions of *Citizens*, *Blockene* borrows an idea from Algorand [21] (modified to make it battery-friendly), where a different random committee of (~2000) *Citizens* is cryptographically chosen to run consensus for each block. Unlike Algorand, *Blockene* exposes the set of committee members a few minutes before their participation: this enables *Blockene* to reduce data and battery cost at *Citizens*. While this may appear to increase the window for a targeted attack on the committee, we discuss in § 4.2 why this is not a serious concern.

To keep storage/communication costs at the *Citizens* low, only *Politicians* store the blockchain and the global state (*i.e.*, key-value pairs), freeing *Citizens* from gossiping all blocks (~50GB/day). *Citizens* only read a small subset of data from *Politicians* (*e.g.*, key-values for transactions for the current block), and write out the new block. Further, because *Politicians* are untrusted, *Citizens* cannot rely on the correct latest values returned by them for, say, a given key. *Blockene* uses a novel technique of *sampling-based Merkle tree read/write* that reduces communication cost while ensuring tolerance to 80% malicious *Politicians*.

When in the committee, *Citizens* reduce their communication cost by not gossiping directly, but through *Politicians*; data written by a *Citizen* gets gossiped among *Politicians*, and interested *Citizens* read from *Politicians*.

As participation in *Blockene* is lightweight, the system needs to protect against Sybil attacks [17]; preventing an adversary from spinning up lots of virtual nodes to get disproportionate voting share. To thwart such attacks, *Blockene* requires the participant identity to be certified by the trusted hardware (TEE) available in most smartphones [6, 11], and enforces that each TEE can have at most one active identity on the blockchain, thus raising the economic cost of participation to the cost of a unique smartphone.

To limit damage that 80% malicious *Politicians* can cause to performance, *Blockene* employs several techniques to restrict their ability to lie. First, we use a technique called *pre-declared commitments* to make some malicious behaviors detectable. Second, to perform gossip among *Politicians* reliably and efficiently despite 80% dishonesty, we introduce a novel technique called *prioritized gossip*. These techniques reduce cost at *Citizens*, enabling *Blockene* to achieve high throughput despite running on smartphones.

We have built a prototype of *Blockene*; the *Citizen* node is implemented as an Android application, and *Politician* node is implemented as a cloud server. We evaluate *Blockene* along various dimensions, and show that it achieves good transaction throughput of 1045 transactions/sec (6.8 MB/min) while ensuring a commit latency of 270s in the 99th percentile. We also demonstrate very little data use (61 MB/day) and battery use (3%/day) at *Citizens*.

The key contributions of this paper are as follows:

- We present the first blockchain system where nodes can participate as first-class members in consensus while running on devices as lightweight as smartphones, supporting high scale of members and high throughput.
- We present a novel split-trust design with a new security model comprised of resource constrained *Citizens* (honest majority) and resource heavy *Politicians* (dishonest majority), and *Citizens* performing validation and consensus by offloading heavy work to untrusted *Politicians* in a verifiable way.
- We make several novel optimizations (e.g., pre-declared commitments, sampling-based Merkle tree read/write, prioritized gossip) that achieve good performance despite 80% malicious *Politicians*.
- With a thorough theoretical analysis, we prove that *Blockene* satisfies *safety, liveness,* and *fairness*.
- We perform a thorough empirical evaluation of this architecture, demonstrating its feasibility as a shared scalable blockchain service.

The rest of the paper is structured as follows: In § 2, we provide a background on blockchains, and discuss existing blockchain architectures in § 3. § 4 provides an overview of *Blockene*, and its threat model, and § 5 presents its design. We discuss optimizations for resource-heavy steps in § 6, present an overview of safety and liveness proofs in § 7, and describe the implementation in § 8. We evaluate *Blockene* in § 9, and conclude (§ 10).

2 Background

In this section, we discuss the key principles and abstractions in a blockchain, and its applications.

2.1 Basic properties

A blockchain is a *distributed ledger* of transactions. Without a trusted authority (*e.g.*, a bank) managing the ledger, a group of mutually untrusted parties collectively validate transactions, and maintain a consistent ledger, provided at least a threshold of participants (*e.g.*, two-thirds) are honest. A blockchain must provide *safety*, *liveness*, and *fairness*. Safety ensures that honest participants have a consistent view of the ledger. Liveness ensures that malicious participants cannot indefinitely stall the blockchain by preventing new block additions. Fairness ensures that all valid transactions submitted to the blockchain get eventually committed.

2.2 Building blocks

A blockchain is a replicated, peer-to-peer distributed system built on the following basic primitives:

Merkle tree for Global State: A key part of a blockchain is the *global state* database that tracks keys and their current values. This global state is managed in a tamper-proof manner, typically using a Merkle tree where the leaf nodes contain the key-value pairs, while each intermediate node contains a hash of the concatenated contents of child nodes. The root is a single hash value that represents the entire state. An update of a key requires recomputation of hashes only along the *path* from that leaf to the root. Given the root, the value of any key can be proved by a path of valid hashes to the root.

Signed transactions: The basic unit of work in a blockchain is a *transaction*. A transaction reads and updates a few keys in the global state (*e.g.*, transfer $1000 from Alice to Bob). To be valid, (a) the transaction must be signed (b) the user signing the transaction must have access to the keys (c) "semantic" integrity must pass (*e.g.*, cannot overspend).

Cryptographic linkage: A blockchain is a list of *blocks*. A block is a list of transactions. The ordering of blocks is ensured by a *cryptographic linkage*; every block embeds the cryptographic hash of the previous block's contents.

Gossip: Participants in a blockchain exchange state with each other in a peer-to-peer fashion. For example, when a new block gets committed to the ledger, it must be sent to other members. This communication happens through multi-hop *gossip*, with eventual consistency.

Consensus Protocol: The key primitive in blockchains is a *distributed consensus* protocol that handles *Byzantine failures* (*e.g.*, PBFT [15], Nakamoto [30], or BBA [21]), as minority of participants could be malicious. Byzantine consensus requires at least 2/3rd participants to be honest, and requires several *rounds* of communication.

3 Comparison with Existing Blockchains

In this section, we present a brief survey of related work on existing blockchain architectures. *Blockene* provides three properties: lightweight resource usage, large scale of participation, and high transaction throughput. We use the same three dimensions to compare *Blockene* with related work.

3.1 Resource usage by member nodes

Existing blockchains span a wide spectrum in resource usage by participating member nodes, depending on the mechanism used for consensus. We first discuss compute cost incurred by members, and then the network and storage cost.

Compute Cost. In terms of compute cost, the most expensive are blockchains based on Nakamoto consensus [30], also referred to as *proof-of-work*; examples are Bitcoin [30] and Ethereum [5]. In Nakamoto consensus, the first member node to solve a compute-intensive cryptographic puzzle is chosen as the winner in committing a new block. Such blockchains therefore require heavy compute resources at member nodes.

In order to address the high compute (and energy) costs of proof-of-work blockchains, two popular alternative architectures have emerged. The first is *consortium blockchains* (*e.g.*, HyperLedger [13]), which, by limiting the blockchain membership to a small number of nodes, can run traditional Byzantine consensus algorithms, instead of the compute-intensive proof-of-work based consensus. The second architecture is *proof-of-stake* blockchains, which tie the voting power of a member node with the amount of money the member node has on the blockchain. Examples of these blockchains are Algorand [21], Ouroboros [14, 22], PeerCoin [23], *etc.*. Inherently, proof-of-stake blockchains target cryptocurrency applications where such a "stake" is meaningful.

Network and Storage cost. While the above two architectures, *i.e.*, consortium blockchains and proof-of-stake blockchains, address the raw compute cost of member nodes, they are still too expensive for smartphones. In particular, they are heavy on network and storage resources, as they require the member nodes *to be always up-to-date* with the "current" state of the blockchain. Given the high transaction rate (1000s of transactions per second) that such blockchains enable, replication of the entire state across member nodes is expensive: at 1000 transactions/sec, the blockchain would commit roughly 9GB per day, which needs to be gossiped across member nodes, resulting in a network cost of roughly 45 GB/day (assuming a gossip fanout of 5 neighbors) that every member node has to incur. Further, such a blockchain would consume terabytes of storage on member nodes, as every member node stores a local copy of the blockchain.

Even blockchains that target smartphones [35] adopt the same philosophy of member nodes staying up to date, and thus incur the network and storage overheads.

Some blockchains address storage cost by *sharding*. OmniLedger [25] is a recent blockchain that allows participants to only store a *shard* of the blockchain. It uses a variant of Byzcoin [24] for fast consensus. RapidChain [37] also uses sharding to reduce storage cost. Both these works scale only to a few thousand participants and also require participants to store a large fraction ($\frac{1}{3}$ or $\frac{1}{16}$) of the entire blockchain.

Lightweight but Incapable Nodes. A class of "lightweight" blockchains adopt an approach of "unequal members": only the first-tier, resource-heavy members participate in consensus and have voting power, while the second-tier members simply serve as read-only query frontends, and do not participate in consensus. In such a model, the "majority-honest" property must be met purely by the heavy nodes, as light nodes do not contribute to security. Not surprisingly, given the limited responsibility, the "light" nodes don't consume much resources. An example of this architecture is the separation between light and heavy nodes in Ethereum [32].

Blockene. In contrast, *Blockene*, achieves lightweight resource usage for *first-class members* that participate in consensus and block validation. Further, unlike Ethereum which depends on honest majority among heavy nodes (only heavy nodes can vote), *Blockene* tolerates up to 80% of the "heavy" nodes (*i.e.*, *Politicians*) being corrupt. Members in *Blockene* require only a smartphone and negligible[2] data transfer (< 60 MB/day, *i.e.*, three orders of magnitude lower) and negligible compute (battery use of <3% per day). It achieves this by enabling member nodes to operate with minimal state needed for committing a particular block, and performing work only a few times a day, *i.e.*, not striving to stay up-to-date always.

3.2 Scale of participation

As the security of a blockchain fundamentally relies on a majority of the participating members being honest, blockchains need to protect against collusion of a large number of participants. Consortium blockchains [13] carefully structure the blockchain for a particular business process, such that members have a shared incentive in the success of the blockchain. It is sometimes infeasible/hard to structure a consortium with the above guarantee; in the philanthropy example, if a small number of members are in control of the blockchain, they may collude to, say, facilitate siphoning of donations meant for the poor. Moreover, a consortium blockchain is intricately tied to a specific business process among a set of entities, resulting in high setup and operational overhead, besides limiting inter-operability.

Another approach to guard against collusion among majority, is to enable large scale participation; by onboarding a large number of participants (say millions), majority-collusion can be made hard and unlikely. Most "public" blockchains such as Bitcoin [30], Ethereum [5], and Algorand [21] enable large-scale participation. Blockene also supports a large number of participants, but unlike most public blockchains today that target cryptocurrencies, *Blockene* is not tied to cryptocurrency (*e.g.*, no proof-of-stake), but enables generic business transactions. Unlike consortium blockchains, *Blockene* can additionally enable real-world scenarios where there is potential for collusion among a small number of members.

[2]Cellular data costs in several countries are much cheaper than in the US [18]; in US/Europe, users are on WiFi/broadband most of the day.

3.3 Transaction throughput

Public blockchains based on proof-of-work are low in throughput (~4-10 transactions/sec). Proof-of-stake based Algorand [21] is the first public blockchain with ~1000 transactions/sec[3] Consortium blockchains, due to low scale of participants and traditional consensus (e.g., PBFT), provide 1000s of transactions/sec. Similar to Algorand, *Blockene* also provides a high transaction throughput. By not being tied to cryptocurrency applications, *Blockene* can serve traditional business applications similar to consortium blockchains.

Blockchain	*Scale of members*	*Trans.* rate	*Cost*	*Incentive needed?*
Public (*e.g.*, Bitcoin)	**Millions**	4-10 /sec.	Huge (PoW)	Yes
Consortium (*e.g.*, [13])	Tens	**1000s /sec.**	High	Yes
Algorand [21]	**Millions**	**1000-2000/sec.**	High	Yes
Blockene	**Millions**	**1045 /sec.**	**Tiny**	**No**

Table 1: Comparison of blockchain architectures.

3.4 Incentives to Participants

Because of high resource cost (compute, network, or storage), existing blockchains need an incentive for participants (*e.g.*, mining coins in cryptocurrencies, or business efficiency in consortiums). Blockchains that depend on such incentives cannot work for applications such as philanthropy (§ 1). To scale without incentives and to enable *altruistic* participation, the cost of participation has to be negligible.

Table 1 compares blockchain architectures along these dimensions. *Blockene* is the first blockchain to achieve all of the above: scale, throughput, and low cost. With low cost, *Blockene* supports real-world use-cases even where participants do not have a direct incentive, but are altruistic to run a background app with negligible battery and data usage.

3.5 Other related work

The committee-based consensus in *Blockene* is heavily inspired by Algorand [21]; Like *Blockene*, Algorand also

[3]Assuming 100-byte transactions and 2.2 MB in 20s, 10MB blocks @750MB/hr.

does not allow forks to occur, and one consistent view of the blockchain is always maintained. There is a tradeoff between Algorand and *Blockene* on the resilience to two kinds of targeted attacks (described in § 4.2 para 1). HoneyBadger [28] is a recent system designed for consortium blockchains with O(100) participants. IOTA [19,20] is another distributed ledger system, but currently relies on a centralized co-ordinator for consensus.

Among proof-of-work-based blockchains, the most closely related work to *Blockene* is Hybrid consensus [31]. Similar to Algorand (and *Blockene*), Hybrid consensus periodically selects a group of participants and does not allow the adversary to corrupt nodes during the "participant selection interval". However it has a long selection interval (of about 1 day) and is also open to the possibility of forks.

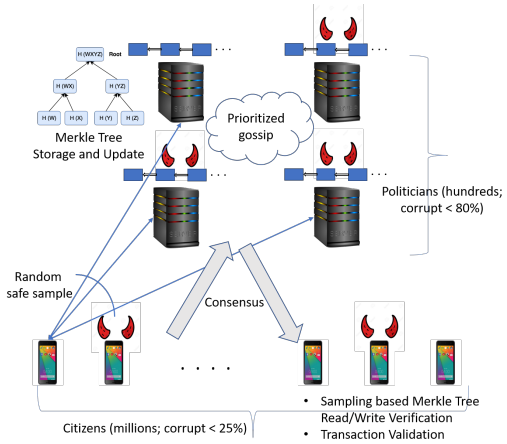


Figure 1: *Blockene*'s architecture

4 Architecture Overview

In this section, we first introduce our two-tier architecture that achieves the three conflicting properties of lightweight resource usage, large scale of participation, and high transaction throughput. We then discuss the threat model of *Blockene*.

4.1 Two-tier Architecture

Blockene employs a novel two-tier architecture with asymmetric trust. This architecture is depicted in Figure 1.

There are two kinds of nodes in *Blockene*: *Citizens* and *Politicians*. *Citizens* are resource-constrained (*i.e.*, run on smartphones), are large in number (millions), and are the only entities having voting power in the system (*i.e.*, participate in consensus). *Politicians* are powerful and run servers (similar to existing blockchains like Algorand), and are lot fewer in number (low hundreds), but they do not have voting power. *Politicians* only execute decisions taken by *Citizens*, and cannot take any decisions on their own.

The low resource usage enables a large number of *Citizens* to participate without incentives, while *Politicians* being few in number, will be run by large entities that have interest in the particular use-case (*e.g.*, in the audited philanthropy case, large donors and foundations).

As *Citizens* participate in consensus, at least two-thirds of *Citizens* are required to be honest, while others can be malicious and collude. This is reasonable as *Blockene* allows millions of *Citizens*, making large-scale corruption hard. However, *Politicians* enjoy much lower trust. *Blockene* only requires 20% of politicians to be honest; the remaining 80% of the politicians can be malicious and collude among themselves, and with one-third malicious *Citizens*.

4.1.1 Offloading work to *Politicians*

Intuitively, given the two-tier architecture, *Citizens* can offload expensive responsibilities such as storage and communication to *Politicians*. However, as 80% of *Politicians* are corrupt, a write made by a *Citizen* could just be dropped by a *Politician* or, a read could return incorrect value. To get useful work done out of *Politicians* *Blockene* uses a novel mechanism of replicated reads and writes. Reads and writes by *Citizens* to *Politicians* happen with a random *safe sample* of *Politicians*. The size of this sample is fixed such that with high probability, at least one *Politician* in the sample is honest (*e.g.*, for a sample size of 25, this probability is $1-(0.8)^{25} = 99.6\%$). *Blockene* is resilient to a small number of *Citizens* (0.4%) picking all dishonest *Politicians*.

4.1.2 Division of responsibilities

We now describe how the *Citizens* and *Politicians* collaborate to perform the various standard blockchain tasks:

Storage: In a traditional blockchain, every participant keeps a replica of the entire blockchain, but *Citizens* in *Blockene* cannot afford to store TBs of data. In *Blockene*, only *Politicians* store the ledger and the global state (*i.e.*, database of key-values § 2). *Citizens* read subsets of this data from *Politicians* as needed. The only state *Citizens* store (and periodically update) is a list of valid *Citizen* identities (§ 5.3).

Transaction Validation: As *Citizens* are the actual participants in consensus, they validate transactions, ensuring that transactions are signed, and have semantic integrity (*e.g.*, no double-spending). To perform validation, *Citizens* read transactions from *Politicians*, and lookup latest values of the keys referenced in them, from the global state with *Politicians*. *Citizens* then propose a block with valid transactions.

Gossip: To ensure that all honest participants agree on the state of the blockchain, participants need to gossip among themselves. However, as discussed in § 3, direct gossip among *Citizens* is expensive. *Blockene* solves this problem by having *Citizens* gossip through *Politicians*. When a *Citizen* needs to broadcast information to other *Citizens*, it sends a message to a *safe sample* of *Politicians*. *Politicians* then gossip data

among themselves; they can afford to do so because they have good network connectivity. Other *Citizens* then perform a replicated read from the *Politicians when they need to, e.g., when they are in the committee*[4]. For gossip through *Politicians*, we need the guarantee that a message that reaches one honest *Politician* always reaches all other honest *Politicians* via gossip, a challenging property when 80% of the *Politicians* are malicious; our custom gossip protocol is described in § 6.1). Thus, we achieve the same semantics as direct gossip among *Citizens*, but with minimal network load on *Citizens*.

Consensus: *Citizens* participate in consensus by performing gossip through *Politicians*. Given the large scale of *Citizens*, all *Citizens* cannot participate in consensus. Instead, we cryptographically select a random *committee of citizens* (roughly 2000 members) for each block (§ 5.2).

4.2 Threat Model

While our threat model is similar to Algorand [21], there is a tradeoff between Algorand and *Blockene* on the resilience to targeted attacks. On one hand, Algorand is based on proof-of-stake, which allows an adversary infinite time to target nodes with higher stake (who will appear in the committee more frequently); *Blockene* avoids this attack, as all *Citizens* have equal votes. On the other hand, Algorand protects the secrecy of the committee members until they perform their role, but *Blockene* exposes their identities a few minutes (1-2 blocks) *before* they participate. To conserve battery, *Citizens* normally poll *Politicians* for current state of the blockchain roughly every 10 blocks (5.2), but when they are going to be in the committee, will poll again shortly (*e.g.*, 1 block) before their expected turn, thus exposing their identity to malicious *Politicians*. This potentially provides a window for a targeted attack (*e.g.*, by bribing the committee: § 4.2.1).

4.2.1 Attack vector of *Citizens*

Bribing attack on *Citizens*: As *Blockene* implicitly exposes the public keys of the committee a few (*e.g.*, 2) minutes in advance, an adversary could in theory perform a targeted attack by bribing a sufficient number of committee members. However, we believe this is not a concern for the following reasons. First, with just the IP address, it is non-trivial for an adversary to "send a message" offering bribe to a *Citizen*, because of carrier-grade NAT [4] and the architecture for push notifications in smartphones ; the existing channel from a malicious *Politician* to the *Citizen* cannot be misused for this, as an untampered *Blockene* app on the *Citizen* will ignore any spurious traffic on that channel. Second, as the committee is randomly chosen every block, pull-based bribing where the *Citizens* (who know of their selection up to 10 blocks in advance - § 5.2) pro-actively reach out to the adversary cannot happen, as that would imply violation of the honesty assumption on *Citizens*, *i.e.*, greater than 70% being honest.

Sybil Attack by *Citizens*: Given the lightweight cost of participation, *Blockene* needs to ensure that an adversary cannot get disproportionate share of voting by spinning up several virtual nodes (*i.e.*, Sybil attacks [17]). A common way of addressing Sybil attacks is *Proof-of-work* which is resource-intensive and does not fit the goals of *Blockene*; another alternative is *Proof-of-stake [21]* where a participant's voting power is proportional to the amount of "stake" (money) on the blockchain, but it is specific to cryptocurrencies.

In *Blockene*, we protect against Sybil attack by exploiting the trusted hardware (TEE) available in smartphones [6, 11], and ensuring that a smartphone can have at most one identity on the blockchain. Thus, *Blockene* imposes an economic cost to participation, *i.e.*, the cost of a smartphone; this is sunk cost *already incurred* in owning the smartphone, but protects against Sybil as each identity is a unique smartphone.

In particular, each TEE has a unique public key that is certified by the platform (Android/iOS) vendor. The TEE can certify an EdDSA public-private keypair generated by an app; this generated public key serves as the identity on *Blockene*. The global state of *Blockene* tracks the set of valid public keys, along with the public key/certificate of the TEE that authorized it. When a transaction for adding a new member is proposed, *Blockene* looks up the TEE public key to see if that TEE (*i.e.*, the same smartphone) already has an identity in *Blockene*; if yes, it rejects the transaction[5]. Thus, every *Citizen* on *Blockene* is tied to a unique smartphone, making it economically infeasible/unattractive for a single entity to get large participation on *Blockene*.

Note that *Blockene* only assumes that every certificate signed by Google/Apple for a TEE public-key corresponds to a unique smartphone. It does not depend on the security of an individual TEE (unlike running the blockchain consensus inside TEE, e.g. SGX [33], that opens up side-channel attacks compromising integrity and security). As a result, the TEE identity can be replaced/combined with other unique identities. In India, one-way-hash of Aadhaar-ID [1, 12] (digitally verifiable, biometric-deduped, 1.2 billion-reach) can be used. Other de-duped IDs (e.g.SSN) augmented with digital verifiability can also be used.

4.2.2 Attack vector of *Politicians*

Dealing with 80% dishonesty among the politicians is one of the main technical challenges in the design of *Blockene*. Malicious behavior by *Politicians* falls under two kinds: detectable and covert. Detectable maliciousness where there is a succinct proof of lying, can be used to improve performance by blacklisting. For example, if a *Politician* is supposed to only send one group of transactions in a round, but there are

[4] Direct gossip among *Citizens* would require *all Citizens* (*i.e.*, including those outside the committee) to participate in gossip of all data.

[5] We can also support replacing the old identity with the new one for the same TEE with appropriate bookkeeping.

two versions signed by the same *Politician*, it is detectable with proof. Covert maliciousness is harder to handle, and is the focus of our techniques. We list broad (non-exhaustive) classes of covert attacks a *Politician* can employ.

Staleness Attack: When a *Citizen* node asks the *Politician* for some state (*e.g.*, the latest committed block), the *Politician* could return a stale block. Such a response would appear to be valid because the old block would also have been signed by a quorum of *Citizens* (§ 5.3).

Split-View Attack: A *Politician* can respond selectively to some *Citizens* and not to others, causing a split in the world-view seen by honest *Citizens*. Worse, a *Politician* can respond with two different values to different subsets of *Citizens*. In a coordinated split-view attack, the malicious *Politicians* could only gossip among themselves, so that no honest *Politician* has a certain data. Malicious *Politicians* can then selectively relay this data only to some *Citizens* (*e.g.*, § 5.5.2).

Drop Attack: A malicious *Politician* may drop data written by a *Citizen* without committing it or gossiping it to other *Politicians*. Similarly on a read, the *Politician* may choose to not respond, even though the *Politician* has the data (§ 4.1.1).

Denial-of-Service Attack: As *Politicians* are powerful servers typically hosted in the cloud, we assume that honest *Politicians* employ standard DoS protection that public clouds offer [2, 3]. For *Citizens*, most ISPs employ carrier-grade NAT to handle the explosion of IP addresses on mobile phones [4], which also provides DoS protection. Malicious *Politicians* can make our gossip protocol more expensive by asking for more data than they need (§ 6.1).

Sybil Attack: An adversary could try pushing the dishonesty fraction of *Politicians* beyond 80% by spinning up several nodes. However, given the small number (say 200), we envision that *Politician* nodes would have an out-of-band registration mechanism (*e.g.*, mapping them to real entities, say one per Fortune-500 company) - robust because only 20% of them need to be honest (unlike *Citizens*).

Blockene protects against both detectable and covert maliciousness of the *Politicians* including the attacks listed above.

5 Design

In this section, we present in more detail how *Citizens* and *Politicians* coordinate on the key steps in *Blockene*.

5.1 System Configuration

We first outline the system configuration for *Blockene*. *Citizens* in *Blockene* run on a smartphone, so we assume that their network bandwidth is low, *i.e.*, 1 MB/s. We choose a block size of 9MB (to amortize fixed cost per block), containing about 90k transactions (~100 bytes each including a 64-byte signature). We assume a network bandwidth of 40 MB/s between *Politicians* (representative of bandwidth in the cloud, *e.g.*, between an Azure and a Google Cloud VM across east-US and west-US). We choose the number of *Politicians* as 200. The work done per block only depends on committee size, so the system scales to millions of *Citizens*.

Transaction originators submit signed transactions to a safe sample or to all *Politicians*, continuously in the background. Transactions can modify keys that the originator has access to. Transactions from the same originator can depend on each other; we preserve their order by tracking a per-originator *nonce* in the global state. In this paper, (without loss of generality) each transaction accesses three keys (debits one key and credits another, third key is nonce). *Politicians* gossip transactions among each other.

5.2 Selecting Committee of *Citizens*

The committee of citizens for validating and signing each block is chosen on the basis of a *VRF (Verifiable Random Function)* [27], inspired by Algorand [21] but with one key modification. Algorand requires each participant to check in each round whether it is chosen in the committee. A *Citizen* on a mobile phone cannot afford to do such frequent checks because waking up the phone every round and communicating would cause significant battery drain. Therefore, instead of computing the VRF on the hash of the previous block ($N-1$), *Blockene* uses the hash of block $N-10$, thus allowing a *Citizen* to wake up once every 10 blocks. Note that this modification still preserves the security guarantee required from VRFs in our threat model. Specifically, for a citizen, the VRF for block N is calculated as $\mathsf{Hash}(\mathsf{Sign}_{\mathsf{sk}}(\mathsf{Hash}(\mathsf{Block}_{N-10})||N))$ where sk is private key known to the citizen. [6] A *Citizen* is in the committee if the VRF has 0's in the last k bits (hence a *Citizen* is part of a committee with probability 2^{-k}; k can be set appropriately). Only the concerned *Citizen* can generate the VRF as it requires its private key, but anyone can verify its validity based on the public key given the signature.

Committee size: The size of the committee needs to balance performance and security. A small committee is good for performance, but for security of consensus protocol, we require that in any committee, at least $2/3$ *Citizens* are honest. As our committee selection is probabilistic, by the Chernoff bound [29], this security requirement cannot be met for very small committee size even if we have $2/3$ honest *Citizens* overall. Committee size increases with the fraction of dishonest *Citizens*. We calibrate this tradeoff to obtain an expected committee size of 2000 with a citizen dishonesty threshold of 25%. The details of these computations appear in the full version [34]. We give an overview below.

Proof overview: We prove several properties about the committee for a block. We call a *Citizen* that participates in a committee as *good* if the *Citizen* is honest and speaks to at least 1 honest *Politician* through m fan-out read/write. Otherwise, we say that the *Citizen* is *bad*. For a configuration with 25% corrupt *Citizens*, 80% corrupt *Politicians*, and $m = 25$,

[6] We use EdDSA signatures. ECDSA uses random number which the adversary can exploit to brute-force itself into the committee.

we show that our committee satisfies the following properties [7]: size of all committees lies in the range [1700..2300] (Lemma 1), every committee has at least 1137 good citizens (Lemma 2), every committee has at least a 2/3 fraction of good citizens (Lemma 3), and no committee has more than 772 bad citizens (Lemma 4).

5.3 Fork-proof Structural Validation

Blockene is designed to prevent forks from occurring. To enable this, each *Citizen* periodically verifies the *structural integrity* of the blockchain to enforce that the chain of hashes and VRFs are consistent and to prevent forks.

Track local state: Each *Citizen locally* remembers the block number N until which the *Citizen* validated the structural integrity of the blockchain, and the hashes of blocks N to $N-9$. In addition, a *Citizen* stores an up to date list of public keys of other valid *Citizens*. The total storage size is <100MB for 1 million *Citizens*.

Chained ID sub-blocks: To enable *Citizens* to efficiently update local state, the public keys of new users added as part of each block B, are tracked in an *ID sub-block (SB)* within B. SBs are chained together by embedding Hash(SB_{i-1}) within SB_i. To aid cheap verification, committee members sign Hash(Hash(B_i), Hash(SB_i), GlobalStateRoot(B_i)).

Incremental Validation: Roughly every 10 blocks (12-15 mins), each *Citizen* performs a `getLedger` call to validate the incremental structural integrity (*i.e.*, from last validation point to the latest state), and to check if it will be in the committee soon (committee for a block N is a function of the hash of block $N-10$). To find the latest block, a *Citizen* queries a safe sample of *Politicians* for the latest block number. It picks the highest number reported by *any Politician*, and asks for proof, *i.e.*, signatures of committee of that block and the corresponding VRFs. Thus, if at least one *Politician* in the safe sample is honest, the *Citizen* will know the latest block hash. If the latest block is greater than $N+10$, it first verifies block $N+10$. Further, it refreshes its set of valid public keys by downloading the chained sub-blocks $SB_{N+1}...SB_{N+10}$ that contain new *Citizens* added in each block, verifying the integrity of SB_i based on the chained hashes.

Cool-off period for new nodes: To prevent a (low-probability) attack where an adversary can manufacture public-private keypairs[8] to increase chances of getting higher malicious fraction for a particular block N, we allow a *Citizen* to be in the committee only $k(=40)$ blocks after the block in which the *Citizen* was added. To verify this as part of VRF checks, a *Citizen*'s local state tracks the block number of "recently" added *Citizens*. This is similar to the "look back parameter" in Algorand [21].

[7] All references to Lemma/Theorem/Algorithm numbers below and rest of this paper refer to those in the full version.

[8] Android TEE API does not allow directly signing with the private key of TEE; instead a keypair is certified by TEE.

Proof overview: Our `getLedger` protocol [34] is used for verifying ledger height $i+10$, given the *Citizen* v has last verified height i, *without an explicit brute-force verification of signatures of all* 10 *blocks*. The algorithm generalizes to verifying any height $i+j$ for $1 \leq j \leq 10$. We show (Lemma 5) that if a *good Citizen* with a verified state for height i invokes the `getLedger` protocol at round $(i+11)$ and accepts, then the *Citizen*'s updated *structural state* is consistent with the blockchain up to height $(i+10)$. Using this, we can show that honest *Citizens* can obtain the consistent structural state of the blockchain, along with all registered public keys, for every round of the protocol (Corollary 2).

5.4 Transaction Validation

Citizens perform the task of verifying signatures of transactions, checking the transaction nonce to detect replay attacks, and verifying semantic correctness of the transaction (*e.g.*, double spending). However, only *Politicians* store the Merkle tree (§ 2.2) of the global state; keeping a large and up to date global state in *Citizens* is unaffordable. To validate a transaction, a *Citizen* must lookup the correct value of keys referenced therein. On commit, the *Citizen* must *update* the Merkle tree with new values from the transaction, and sign the new Merkle root. The challenge lies in doing so correctly given untrusted *Politicians*.

The Merkle root (along with block number) is signed by the committee of the previous block, so the *Politician* cannot lie about the Merkle root. Once the *Citizen* learns the latest block number (§ 5.3), it learns the correct Merkle root as well. To verify a value returned for a key, *Citizen* asks the *Politician* to send the *challenge path* for this key, i.e., all the sibling nodes (hashes) along the path from the leaf to the root. This enables the *Citizen* to reconstruct the Merkle path and match the root hash with the signed Merkle root. By security of hashes, the *Politician* cannot present spurious challenge paths that verify. In a tree with 1 billion key-value pairs, the challenge path would contain 30 hashes.

Update of keys in the Merkle tree follows a similar protocol. The *Citizen* could build a partial Merkle tree with the new values at the leaves, and compute the new Merkle root. Both the read and update paths mentioned above are expensive, and we optimize them in § 6.

5.5 Block Proposal

Like in any blockchain, committee members can propose a new block for committing to the blockchain.

5.5.1 Pick winning proposer

For efficiency, we allow only a subset of committee members called *proposers* to actually propose a block, based on the *VRF* of the *Citizen*. For this selection, we use an additional

VRF that is based on the hash of the previous block $N-1$ (instead of $N-10$); only committee members who have the last k' bits of the additional VRF set to zero can propose a block, and the winner is the one with the least VRF. Using the previous block hash in this VRF ensures that the adversary does not know about the proposers until the last minute (similar to Algorand) thus preventing a targeted attack on the proposers. Any committee member can consistently determine the winning VRF among the proposers. All proposers upload their block to *Politicians* and other committee members download the block of the winning proposer.

5.5.2 Pre-declared commitments

The upload of the proposed block by a proposer needs to be done to a safe sample of 25 *Politicians*. In *Blockene*, as the blocks are ∼9MB in size, assuming 1MB/s bandwidth at mobile nodes, this would take 225 sec. To optimize this step, we make the transaction selection process deterministic, so that *any Citizen can reconstruct what the original proposer would have done*, without the proposer explicitly uploading the full block. Determinism is challenging, however, because the 80% malicious *Politicians* can send different transactions to different *Citizens*. Our technique of *pre-declared commitments* to transactions addresses this.

1. Freeze Transactions. At the start of block N, each *Politician freezes* the exact set of transactions it will send to *Citizens* reading from it. It does so by creating a *tx_pool*, which includes a set of (about 2000) transactions, and then generates a *commitment* which is a signed hash of the *tx_pool* along with the block number[9]. Malicious *Politicians* are forced to issue only one commitment for a given block N, because two signed commitments from a *Politician* is a proof of malicious behavior, and can be used for efficient blacklisting; *Citizens* then drop all commitments from that *Politician* in the same round. Intuitively, with frozen commitments, a *Citizen* proposing a block, need not upload the full block, but only a *digest* with the commitments that went into the block, and other *Citizens* can reconstruct that block by downloading the *tx_pools* for those commitments from *Politicians*.

2. Ensure that *enough* honest citizens *have* commitments. A malicious *Politician* can respond with its *tx_pool* only to a subset of *Citizens*, and refuse to respond to others; thus, a *tx_pool* committed in the proposed block may not be readable by all honest *Citizens*, thus thwarting consensus. To address this, we perform three steps. First, we limit the exact set of *Politicians* from whom to pull transactions for a given block to a randomly chosen set of 45 politicians based on the hash of the block number and hash of previous block. Instead of reading *tx_pools* from a random safe sample, a *Citizen* reads from these 45 designated *Politicians* for a block. Second, the *Citizen* uploads a *witness list* to a safe sample of *Politicians*; the witness list contains the list of *tx_pools* the *Citizen* was able to successfully download. The witness list of all *Citizens* gets gossiped between *Politicians*. Third, the proposer reads the witness list of all other *Citizens*, and picks only commitments whose *tx_pools were successfully downloaded by at least a threshold number of Citizens*. This threshold is fixed to be $\tilde{n_b}+\Delta$, where $\tilde{n_b}$ is the maximum number of malicious nodes in any committee (computed to be 772, from Lemma 4), and Δ is chosen to be 350. Intuitively, all commitments (and *tx_pools*) that pass this condition are available with at least Δ *honest Citizens*. As 20% of *Politicians* are honest, in expectation, at least 9 out of the 45 commitments will pass this test.

3. Ensure that *all* honest citizens *get* commitments. The commitments available with at least Δ honest *Citizens* now need to be propagated to all honest *Citizens*. Each *Citizen* in Step 4, *re-uploads* 5 random *tx_pools* it has, to 1 random *Politician*. This ensures that (with high probability) each *tx_pool* (including those from malicious *Politicians*) that belongs to at least Δ honest *Citizens* reaches at least one honest *Politician* (who then gossips it to other honest politicians). Thus, other honest *Citizens* can successfully download that *tx_pool* (by querying a safe sample of politicians), preventing a *split-view* attack by malicious *Politicians*.

4. Handle malicious proposer. When the winner of block proposal is a malicious *Citizen*, it need not respect the witness list criteria, and can pick a commitment whose *tx_pool* is known to very few *Citizens*. This attack is possible only when consensus outputs the block proposed by this malicious proposer, so we can argue that at least $1/3$ honest *Citizens had* all *tx_pools* at the beginning of the consensus. To ensure that all honest *Citizens* are able to download all required *tx_pools*, a second re-upload of randomly chosen *tx_pools* happens (step 9), now including the downloaded *tx_pools* from previous step. Formal proofs capturing the guarantees provided by these re-uploads needed to prove security of our system are presented in Lemmas 10 and 11.

5.6 Block Commit Protocol

The main operation in a blockchain is adding a new block to the blockchain. We list below the key steps in the process of committing block N. The protocol for block N starts once the previous block $N-1$ gathers a threshold number of signatures (set to 850 in our case, §E.1 [34]) from the committee members for block $N-1$.

1. A new committee of *Citizens* is chosen for block N (using Hash of block $N-10$), denoted by C^N. The *Citizens* in C^N keep polling for the latest committed block number, and start the protocol once that number is $N-1$.
2. Each *Citizen* C_i^N in C^N downloads *tx_pools* & commitments from $\rho = 45$ designated *Politicians* for the block.

[9] To reduce overlap of transactions across *tx_pools* from multiple *Politicians* (which would reduce the unique transactions in the final block), transactions are deterministically partitioned across *Politicians* using a hash on transaction identifier and round number. Given a *tx_pool* and commitment, it is easy to detect/ blacklist a *Politician* that doesn't follow this.

3. Each C_i^N uploads a signed witness list with the commitments it downloaded, to a safe sample of *Politicians*.
4. Each *Citizen* C_i^N picks 5 random *tx_pool* it has, and re-uploads them to 1 random *Politician*.
5. Each *proposer* in C^N downloads all witness lists of C^N from a safe sample of *Politicians*, and picks commitments with at least a threshold (1122) of votes (§ 5.5.2). Then, it makes a *block proposal* with those commitments, along with its VRF to prove proposer eligibility.
6. *Politicians* gossip on block proposals/VRFs and on the *tx_pools* that were re-uploaded by *Citizens*.
7. Each *Citizen* C_i^N tries to download missing *tx_pools* in step 2 from safe sample of *Politicians*, relying on the re-upload (Step 4) by other *Citizens*.
8. Each C_i^N reads the VRFs of all proposers in C^N from a safe sample of *Politicians*, and picks the lowest correct VRF as the *local winner*. If C_i^N already has all *tx_pools* in the winning proposal, it enters consensus with that set of commitments, otherwise, NULL.
9. Each *Citizen* C_i^N performs a second re-upload of 10 random *tx_pools* it has to 1 random *Politician*.
10. *Citizens* in C^N run a consensus protocol (§ 5.6.1) with gossip through *Politicians*, where each C_i^N's vote is decided in Step 8. At the end, all honest *Citizens* either agree on same set of commitments or an empty block. C_i^N downloads the *tx_pools* missing w.r.t. the output of consensus from safe sample of *Politicians*.
11. Each *Citizen* C_i^N performs transaction validation by downloading challenge paths for all keys from *Politicians* (§ 5.4) and drops transactions that fail validation.
12. Based on valid transactions (Step 11), each C_i^N creates a block, computes the new Merkle root of the global state using updated values of keys and signs the block hash and new Merkle root, along with block number N. It uploads the block hash, new Merkle root, and this signature to a safe sample of *Politicians*.
13. When more than a threshold number of signatures have accumulated for block N, block $N+1$ starts.

Our complete protocol description can be found in Algorithm 4. We give an overview of various properties of *Blockene*, i.e., safety, liveness and fairness, in § 7.

5.6.1 Consensus Protocol

For consensus (Step 10), we use the Byzantine Agreement (BA) algorithm for string consensus (that is based on [36]) which calls upon the bit consensus algorithm BBA [26] in a black-box manner. These are the same consensus algorithms used by Algorand. *Citizens* enter the consensus protocol with list of commitments in local winning block, as input. Two scenarios are relevant here. If the winning proposer (*i.e.*, the one with the lowest VRF) was honest, which would happen at least two-thirds of the time, all honest *Citizens* in the committee would enter consensus with this proposal except with small probability (Lemma 10), and the protocol will terminate in 5 rounds. However, if the winning proposer was malicious, it can collude with malicious *Politicians* to partition the view of honest *Citizens*. In general, the consensus protocol would take an expected 11 rounds [21].

6 Optimizations

In this section, we present two key optimizations crucial to achieving high transaction throughput in *Blockene*.

6.1 Prioritized Gossip

Problem. The guarantee we require in *Blockene* is that if one honest *Politician* has a message, *all* honest *Politicians* receive the message. Because of the high fraction of dishonesty among *Politicians*, standard multi-hop gossip with a small number of neighbors (*e.g.*, 10) cannot provide this guarantee, because there is a non-trivial probability that all of them were dishonest, and drop the message. Hence the safe thing to do is a full broadcast to all other *Politicians*, which is expensive; when *Politicians* need to gossip *tx_pools* that were re-uploaded by *Citizens* in the committee, each *Politician* may have up to 45 *tx_pools* to gossip; with full broadcast, it would send $0.2MB * 45 * 200 = 1.8GB$ which would take 45 seconds in the critical path (@40MB/s).

Key idea. We leverage the fact that messages being gossiped by the different *Politicians* have a high overlap; each *Politician* has a subset of the same 45 *tx_pools* as *Citizens* pick a random *Politician* to re-upload a subset of *tx_pools*. Moreover, given the nature of re-upload, in expectation, any *Politician* would be missing only a few *tx_pools*, and honest *Politicians* wouldn't lie about state.

1. Handshake. Each *Politician* asks recipients B_i which *tx_pools* they already have, and send only the missing ones. While this works with honest *Politicians*, the 80% malicious ones could always lie that don't have any, to cause a higher load/latency on the system.

2. Selfish gossip. As malicious *Politicians* can lie that they have no *tx_pools*, we assign a soft-penalty to *Politicians* that miss a lot of *tx_pools*. Each sender *Politician A* favors the peer *B* that has the maximum number of *tx_pools* that *A* needs. In each round, *A* sends a *tx_pool* to *B*, and receives one in return. Given the random re-uploads by *Citizens*, each honest *Politician* would be missing only a small number of *tx_pools*, and hence would get prioritized. The list of what *B* has to offer keeps getting updated as *B* gets *tx_pools* from other peers; note that this list can only grow, not shrink.

3. Incentivize frugal nodes. Selfish gossip loses its ability to discriminate between honest and malicious recipients, once the sender receives all *tx_pools*. To address this, after getting all *tx_pools*, the sender changes its priority function for destinations B_i to be *the number of tx_pools that* B_i *claims to have*;

thus honest nodes which will have large fraction of *tx_pools* are favored. Again, the list of *tx_pools* that B advertises can only grow, not shrink, as shrinking would mean that B lied. Further, each honest B_i requests its missing chunk from at most $k = 5$ peers simultaneously; $k = 1$ will be data-frugal, but incur high latency if the peer dishonestly delays response.

6.2 Sampling-based Merkle Tree Read/Write

Problem. The Merkle tree validation in Step 11 is expensive. In a 1-billion node Merkle-tree (30-levels deep), a challenge path is 300 bytes (10-byte hashes); downloading 270K challenge paths is 81 MB ($\tilde{8}1$ sec latency) ignoring compression. The compute at *Citizens* is also high (total 16.2 million hash computations for challenge path verification during read and for computing new root post update).

Key idea. We offload most of this work to *Politicians*, in a verifiable manner. Since the Merkle tree validation is done after the conclusion of the consensus run using gossip through the *Politicians*, *Politicians* know the *tx_pools* that are considered for constructing the block. Hence, all *Citizens* in committee and *Politicians* know the keys whose values need to be read and updated. We first discuss the optimization for reading values correctly from the Merkle tree.

1. Get Values. Each *Citizen* gets just the values for all 270K keys (no challenge path, 1 MB instead of 81 MB) from one *Politician*, and then asks a safe sample of *Politicians* whether those values were correct. As at least one of these *Politicians* is honest, it alerts the *Citizen* to incorrect values through an *exception list*. The *Politician* can "prove" an incorrect value by providing a challenge path from the signed Merkle root that indicates a different value for the key.

2. Spot-checks. If many values were wrong, the exception list would be quite large and eat into the savings. To avoid this, *Citizen* picks a small random subset of $k' = 4500$ keys to initially spot-check using the challenge paths. If the spot-checks pass for a sufficiently large k', a *Politician* could have lied only for a small number (200) of keys (except with small probability). Thus, the extra spot-checks bound the size of the exception list (Lemma 6).

3. Exception list protocol. To *cross-verify* the values with a safe sample of *Politicians*, the *Citizen* deterministically puts these values into buckets (2000) and uploads the hashes of these buckets. When a *Politician* notices a mismatch for a bucket, it sends the bucket index and the correct values for all keys in that bucket. *Citizen* gets challenge paths only for keys that disagree (from first *Politician*). Our spot-checks ensure that only a small number of buckets can mismatch.

Corner case. Even after doing the above, there is a small probability ($< 2^{-10}$) that a *Citizen* may obtain an incorrect value; we count such *Citizen* nodes as malicious and account appropriately (Lemma 7). The full protocol and all proofs are provided in Algorithm 2.

Writes: Updating the Merkle tree is a trickier problem. Due to lack of old challenge paths for the all keys being updated, the *Citizen* cannot construct the root of the updated Merkle tree T'. We solve this problem by making the *Politicians* compute T', but now the *Citizen* must verify that the *Politicians* performed the computation correctly, i.e., T' is consistent with the new values of updated keys and old tree T for unmodified keys. We achieve this by breaking T' at a level called the *frontier level* (the nodes at this level are frontier nodes). *Citizens* obtain the values of the frontier nodes of T' from a safe sample of the *Politicians*. The *Citizens* then run a spot checking algorithm - they pick a random subset of frontier nodes and ask a *Politician* to prove the correctness of that frontier node. Next, *Citizens* create exception lists with the help of the rest of the selected *Politicians*. This list denotes which frontier nodes are incorrect with the *Citizen*. The *Citizen* then proceeds to sequentially correct the incorrect frontier nodes and then finally compute the correct root of T' from the frontier nodes.

Proof Overview: In the full paper [34], we prove (in Lemma 6) that for a *good Citizen*, after successfully spot-checking only μ fraction of key-values, only (a small number of) τ values are incorrect with probability $1 - \varepsilon_1$ (here, μ, τ and ε_1 are appropriately chosen parameters). Moreover, these values will get corrected by processing exception lists of size at most τ. Hence, a good *Citizen* gets correct values with probability $1 - \varepsilon_1$ (Corollary 3). We pick our parameters (Lemma 7) such that at most 18 *good Citizens* will obtain incorrect values during read, and account for these 18, by counting them as *bad Citizens* in the committee. In the write protocol, we can show that the sizes of exception lists can be bounded (Lemma 8) and that no more than 18 *Citizens* accept an incorrectly updated Merkle tree T' (Lemma 9), which we once again factor in to the set of bad *Citizens*. We additionally also show that our algorithms are between $3 - 18\times$ more communication efficient and between $10 - 66\times$ computationally faster than the naive algorithm for global state read/write.

7 Proofs of Safety, Liveness, and Fairness

In this section, we provide a brief overview of the proofs in the full paper [34] for the safety, liveness, and fairness guarantees of *Blockene*.

A committee round N ends when a new block gets signed and committed by a threshold number (T^*), of committee members for N. T^* will be set to be 850 (done taking into account maximum number of bad citizens in any committee as well as the 36 good citizens who might have read/written an incorrect global state).

First, we show (in Lemma 10) that for a block, if a good *Citizen* is the winning proposer, then (except with bounded constant probability) all good *Citizens* will output the proposal of this *Citizen* as the output of the consensus protocol. In Lemma 11, we show that, on the contrary, if a malicious *Citizen* is the winning proposer and the consensus results

in a non-null value, then all good *Citizens* will be able to download the transactions committed in the proposal. Using Lemmas 7 and 9 (see Proof Overview of § 6) , we then show (Lemma 12) that at the end of the block commit protocol all, except 36, good citizens will sign the same block hash and new global state root and that the new block is consistent with the entire blockchain and global state. Now, using Lemma 12, safety (i.e. all honest *Citizens* agree upon all committed blocks and all blocks are consistent with a correct sequence of transactions) follows via an inductive argument. Next, to argue liveness (that adversarial entities cannot indefinitely stall the system and that the empty-block probability is bounded by a small constant), we use Lemmas 12 and 10.

Additionally, we also prove bounds on throughput in Lemma 13 (in expectation, committed blocks have a threshold number of transactions in them) and fairness in Lemma 14 (all valid transactions will eventually be committed).

8 Implementation

We have built a prototype of *Blockene*, that is spread across two components, *Citizen* nodes and *Politician* nodes.

8.1 *Citizen* nodes

The *Citizen* node is implemented as an Android app on SDK v23 and has 10,200 lines of code. It is built to optimize battery use and runs as a background app, without user involvement after initial setup. The application caters to two main phases of the protocol that a Citizen participates in: passive and active. In the passive phase, a service using JobScheduler [9] periodically polls *Politicians* for `getLedger` calls. In the active phase, when the *Citizen* is part of a committee, the application runs the steps of the protocol, handling failures, timeouts and retries to deal with corrupt *Politicians*. The implementation for the active phase uses a multi-threaded event-driven model and is built on top of EventBus to parallelize and pipeline network and compute intensive crypto tasks such as signature validation.

8.2 *Politician* nodes

The *Politician* node is implemented in C++ (11K lines of code). The implementation scales to load from thousands of *Citizens*, and handles bursty load during gossip. Given the state-machine nature of the protocol, we have built it on top of the convenient C-Actor-Framework [16], which is based on "actors" that transition the state of the *Politician* through the steps of the protocol. For instance, the BBA actor, apart from storing and serving the votes that *Citizens* submit, also reads the votes to determine the result of consensus. Based on this, it emits an event to build the updated Merkle tree.

For the global state, we have built a SparseMerkleTree (SMT), where the leaf index is deterministically computed using the SHA256 of the key. Since the tree is of bounded depth, we allow for (a small number of) collisions in the leaf node. The challenge path of any key includes all the collisions co-located with this key, so the leaf hash can be computed. To prevent targeted flooding of a single leaf node, we reject key additions that take a leaf node beyond a threshold, forcing the transaction originator to use a different key. We also implement a DeltaMerkleTree, which allows us to efficiently create an updated version of the SMT using memory proportional only to the touched keys.

Our gossip implementation does simple broadcast for regular messages, and runs a stateful protocol for *tx_pool* gossip. We segregate these messages into different ports/queues so the bursty gossip messages are isolated from small messages (e.g., BBA votes) that are broadcast. To prevent malicious *Citizens* from flooding an honest *Politician* with the responsibility of gossiping their writes, we limit the set of *Politicians* for a *Citizen* to be deterministic based on its VRF. *Politicians* do not gossip messages from non-conforming *Citizens*.

9 Evaluation

We evaluate our *Blockene* prototype under several dimensions. The main questions we answer in our evaluation are:

- What throughput and latency does *Blockene* provide?
- How well does *Blockene* handle malicious behaviors?
- Are the optimizations on Merkle tree & gossip useful?
- What is the load on *Citizen* nodes (battery/data usage)?

9.1 Experimental setup

In our experiments, we use a setup with 2000 *Citizen* nodes and 200 *Politician* nodes. *Citizen* nodes are 1-core VMs on Azure with a Xeon E5-2673, 2GB of RAM, and are spread across three geographic regions across WAN: 700 VMs in SouthCentralUS, 600 VMs in WestUS, and 700 VMs in EastUS. Each *Citizen* runs an Android 7.1 image, and is rate-limited to 1MB/s network upload and download. *Politician* nodes run on 8-core Azure VMs with a Xeon E5-2673, 32 GB of RAM, and are spread as 100 VMs each in EastUS and WestUS. They are rate limited to 40MB/s network bandwidth. Given the random safe sampling, the *Citizen-Politician* communication spans across WAN regions. Similarly, the gossip between *Politicians* happens across WAN regions. As our committee size is 2000, every *Citizen* is in the committee for every block. With a higher number of *Citizens*, say 1 million, a particular *Citizen* will be in the committee only once every 500 blocks. Except the per-*Citizen* load, the system performance is independent of the total number of *Citizens* and is just a function of committee size, so the numbers are representative of a large setup.

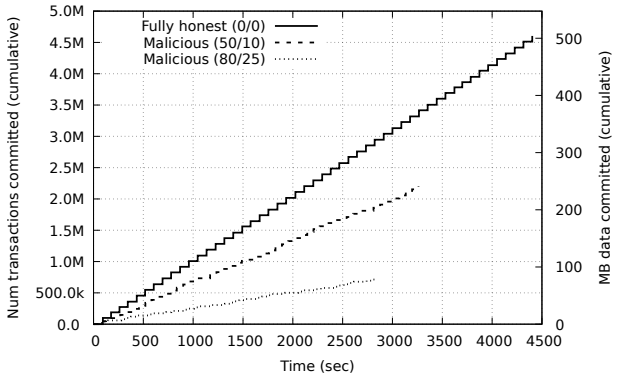


Figure 2: Throughput of *Blockene* under various configs. In 50/10, 50% *Politicians* & 10% *Citizens* are malicious.

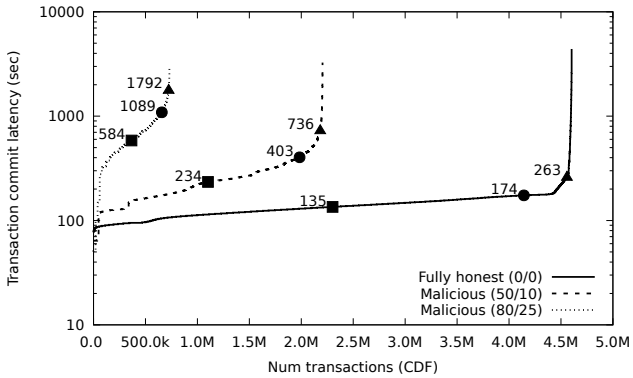


Figure 3: Transaction Latency under different malicious configs. Dots show 50th, 90th, 99th percentiles.

9.2 Transaction Throughput and Latency

Figure 2 shows the timeline of block commits in *Blockene* under fully honest and malicious configurations, for 50 consecutive blocks. In the fully honest (0/0) case, 4.6 million transactions get committed in 4403 seconds, corresponding to a throughput of 1045 transactions per second, or 114 KB/s.

Citizen dishonesty	*Politician* dishonesty		
	0%	50%	80%
0%	1045	757	390
10%	969	675	339
25%	813	553	257

Table 2: **Transaction throughput under malicious configs.**

We also evaluate *Blockene* under malicious behaviors of both *Citizens* and *Politicians*. We denote our malicious configurations in the format P/C, where P is the fraction of malicious *Politicians*, and C is the fraction of malicious *Citizens*. With our choice of parameters (*e.g.*, committee size), *Blockene* is guaranteed to ensure safety in the presence of up to 80% malicious *Politicians* and 25% malicious *Citizens*. However,

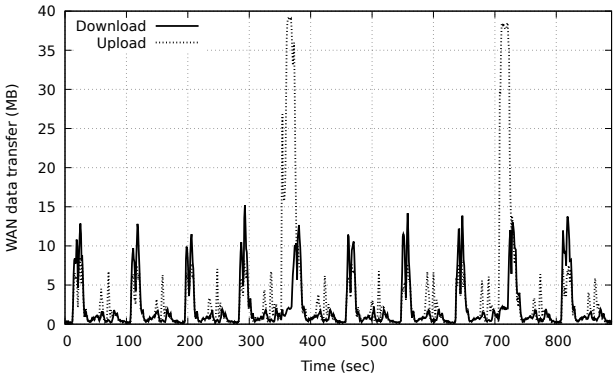


Figure 4: Network usage at a *Politician* node.

performance can be affected because of adverserial behavior. A malicious *Citizen* in these experiments attacks in two ways (a) force an empty block by colluding with malicious *Politicians* and proposes a block with *tx_pools* that only malicious *Politicians* have. Honest *Citizens* therefore cannot download that commitment and will vote for an empty block; (b) forces additional rounds in the BBA consensus protocol by manipulating its votes. A malicious *Politician* attacks in two ways: (a) fails to give out transaction commitments, making a subset of the 45 *tx_pools* empty, potentially causing a smaller block to be committed (b) manipulates gossip by acting as sink holes and asking for same chunks from multiple peers. As Figure 2 shows, *Blockene* is quite robust to a range of malicious behaviors, and gracefully degrades in performance. With 80% dishonest *Politicians*, the effective *tx_pools* reduce to 9 out of 45, resulting in the block having only 18K transactions instead of 90K. Malicious *Citizens* cause a performance hit (empty blocks + BBA rounds) when they get chosen as the proposer (*i.e.*, highest VRF); Table 2 shows the throughput under more configurations of malicious behaviors.

Figure 3 shows the CDF of transaction latencies of the system under different configurations, demonstrating fairness across transactions. In the fully honest case (0/0), *Blockene* ensures a median latency of 135s and a 99th%-ile latency of 263s. Under the two malicious configurations: 50/10 and 80/20, latencies are higher as expected.

9.3 Timeline of *Citizens* and *Politicians*

Figure 4 shows the network load at a typical *Politician* node during 10 blocks (each of the repetitive patterns is a block). The two large spikes in uploaded data correspond to rounds where this *Politician* was one of the 45 chosen to provide *tx_pools*. For each block, there are two small spikes of transmitted data; the first spike corresponds to gossip of *tx_pools* through prioritized gossip, and the second spike is due to gossip of votes from *Citizens* in the BBA consensus.

We also show the breakup of the 89-sec block latency by

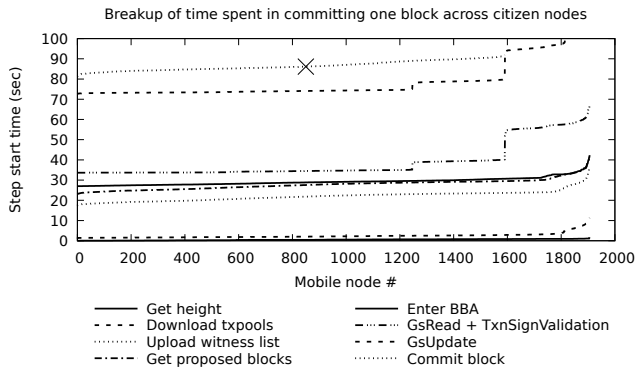


Figure 5: Breakup of time spent at *Citizen* nodes for a single block commit. Cross indicates block commit.

plotting the time taken in *Citizen* nodes during a typical block. Figure 5 shows the progress of the 2000 *Citizen* nodes during one of the blocks, separating out the key phases of the protocol; the bulk of the time goes in the transaction validation phase, and in fetching *tx_pools* from *Politicians*.

9.4 Impact of Optimizations

We now evaluate the prioritized gossip and the sampling-based Merkle tree optimizations. For gossip, we consider how much upload/download each *Politician* incurs before *all other honest Politicians* get all the *tx_pools*. For example, in the 0/0 case, we have 10K data points (across 50 blocks and 200 *Politicians* each). Across these samples, we plot the 50th, 90th, and 99th percentiles. The malicious strategy we model in the 80/25 case is where only the bare minimum number of honest *Citizens* have *tx_pools* of malicious *Politicians* (Δ from § 5.5.2) and all malicious *Politicians* ask for the full set of *tx_pools* from all honest nodes. As Table 3 shows, the network load of prioritized gossip is robust to dishonest behavior. Even in the malicious setting, the data transmitted is quite small before all honest *Politicians* get all *tx_pools*.

Table 4 compares the performance of our sampling based Merkle-tree reads and updates, with the simple solution of downloading challenge paths for all keys referenced in the block. The simple solution incurs much higher network cost (the numbers are after gRPC compression), and a significant compute cost at the *Citizen*. With our optimization, the network cost drops by 10.8× while the CPU cost drops by nearly 31×, thus significantly improving transaction throughput.

9.5 Load on *Citizens*

Finally, we evaluate the load at *Citizen* nodes due to running *Blockene*. The two metrics of interest are battery usage and data usage. To get these metrics, we run an actual Android phone (a OnePlus 5) with the *Citizen* app, as part of the committee along with the 2000 committee members on VMs,

Config	Percentile	Upload (MB)	Download (MB)	Time (sec)
0/0	50	23.1	22.4	3.6
0/0	90	30.5	27.5	4.8
0/0	99	36.7	30.1	5.2
80/25	50	35.4	23.8	3.5
80/25	90	47.6	27.6	4.1
80/25	99	53.4	28.9	4.5

Table 3: **Cost of gossip per honest *Politician* before all honest *Politicians* receive all *tx_pools*.**

Config	Upload (MB)	Download (MB)	Compute (s)
Naive: GS Read	0	56.16	93.5
Naive: GS Update	0	0	93.5
Optimized: GS Read	0.55	1.6	1.0
Optimized: GS Update	0.01	3	5.88

Table 4: **Performance of Global State Read & Write.**

and measure battery use. After being in the committee for 5 blocks, the battery drain was ~3%. The total network traffic incurred by a *Citizen* for a single block was 19.5 MB.

Now, we can extrapolate the daily cost based on the per-block cost and the number of times a single *Citizen* is expected to be in the committee. With 1 million *Citizens*, a *Citizen* will participate roughly every 500 blocks, which at our block latency of ~90s, translates to 2 times per day. Thus, the expected battery use is < 2% per day, and the data use is ~40MB/day. In addition, we also measured on the same OnePlus5 that waking up the phone every 10 minutes and performing `getLedger` costs about 0.9% battery and 21MB data download. Waking up every 5 minutes costs 1.7% battery and 42MB data download. With a total of 3% battery usage and 61MB data/day, a user running the *Blockene* app will hardly notice it running.

10 Conclusion

By enabling, for the first time, a high-throughput blockchain where members perform block validation and consensus on smartphones at negligible resource usage, *Blockene* opens up a much larger class of real-world applications to benefit from the security and decentralized nature of blockchains. With a novel architecture, and several new techniques coupled with a careful security reasoning, *Blockene* is able to simultaneously provide three conflicting properties: large scale of participation, high transaction throughput, and low resource usage at member nodes.

Acknowledgements

We thank our shepherd Nickolai Zeldovich and the anonymous reviewers for their valuable suggestions and feedback. We also thank Ankush Jain, Sriram Rajamani, Bill Thies, Jacki O'Neill, and Rashmi.K.Y for their support.

References

[1] Aadhaar identity ecosystem. In https://uidai.gov.in/aadhaar-eco-system/authentication-ecosystem.html.

[2] AWS Shield: Managed DDoS Protection. In https://aws.amazon.com/shield/.

[3] Azure DDoS Protection. In https://azure.microsoft.com/en-in/services/ddos-protection/.

[4] Carrier-grade NAT: Wikipedia. In https://en.wikipedia.org/wiki/Carrier-grade_NAT.

[5] Ethereum blockchain. In https://www.ethereum.org/.

[6] Apple Platform Security: Secure Enclaves Overview. In https://support.apple.com/en-in/guide/security/sec59b0b31ff/web, 2017.

[7] Giving in Europe country reports available. In https://ernop.eu/giving-in-europe-launched-at-spring-of-philanthropy/, 2017.

[8] India Philanthropy Report 2017. In https://www.bain.com/insights/india-philanthropy-report-2017/, 2017.

[9] Android Docs: JobScheduler. In https://developer.android.com/reference/android/app/job/JobScheduler, 2018.

[10] Charitable Giving Statistics. Americans gave $410 billion to charities in 2017. In https://nonprofitssource.com/online-giving-statistics, 2018.

[11] Android Keystore System: Hardware Security module. In https://developer.android.com/training/articles/keystore#HardwareSecurityModule, 2019.

[12] Ronald Abraham, Elizabeth S Bennett, Noopur Sen, and Neil Buddy Shah. State of aadhaar report 2016-17. *IDinsight. Available at: http://stateofaadhaar. in*, 2017.

[13] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[14] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 913–930, 2018.

[15] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[16] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!*, pages 87–96. ACM, Oct. 2013.

[17] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[18] Forbes. The cost of mobile internet around the world. In https://blogs-images.forbes.com/niallmccarthy/files/2019/03/20190305_Data_Cost.jpg, 2019.

[19] IOTA Foundation. Differences between the tangle and blockchain. In https://docs.iota.org/docs/getting-started/1.1/the-tangle/tangle-vs-blockchain.

[20] IOTA Foundation. Tangle: The coordinator. In https://docs.iota.org/docs/getting-started/1.1/the-tangle/the-coordinator.

[21] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[22] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

[23] S King and S Nadal. Peercoin–secure & sustainable cryptocoin. https://www.peercoin.net/whitepapers/peercoin-paper.pdf, 2012.

[24] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 279–296, 2016.

[25] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 583–598, 2018.

[26] Silvio Micali. Byzantine agreement, made trivial. In https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Distributed%20Computation/BYZANTYNE%20AGREEMENT%20MADE%20TRIVIAL.pdf, 2018.

[27] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 120–130, 1999.

[28] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42, 2016.

[29] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[30] Satoshi Nakamoto. A peer-to-peer electronic cash system. In https://bitcoin.org/bitcoin.pdf, 2008.

[31] Rafael Pass and Elaine Shi. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 39:1–39:16, 2017.

[32] Gary Rong and Felix Lange. Light ethereum sub-protocol (les). In https://github.com/ethereum/devp2p/blob/master/caps/les.md, 2019.

[33] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. Ccf: A framework for building confidential verifiable replicated services. Technical report, Technical Report MSR-TR-2019-16, Microsoft, 2019.

[34] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. Blockene: A high-throughput blockchain over mobile devices. *arXiv preprint arXiv:2010.07277*, 2020.

[35] Kongrath Suankaewmanee, Dinh Thai Hoang, Dusit Niyato, Suttinee Sawadsitang, Ping Wang, and Zhu Han. Performance analysis and application of mobile blockchain. In *2018 international conference on computing, networking and communications (ICNC)*, pages 642–646. IEEE, 2018.

[36] Russell Turpin and Brian A. Coan. Extending binary byzantine agreement to multivalued byzantine agreement. *Inf. Process. Lett.*, 18(2):73–76, 1984.

[37] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 931–948, 2018.

Tolerating Slowdowns in Replicated State Machines using Copilots

Khiem Ngo*, Siddhartha Sen†, Wyatt Lloyd*
*Princeton University, †Microsoft Research

Abstract

Replicated state machines are linearizable, fault-tolerant groups of replicas that are coordinated using a consensus algorithm. Copilot replication is the first 1-slowdown-tolerant consensus protocol: it delivers normal latency despite the slowdown of any 1 replica. Copilot uses two distinguished replicas—the pilot and copilot—to proactively add redundancy to all stages of processing a client's command. Copilot uses dependencies and deduplication to resolve potentially differing orderings proposed by the pilots. To avoid dependencies leading to either pilot being able to slow down the group, Copilot uses fast takeovers that allow a fast pilot to complete the ongoing work of a slow pilot. Copilot includes two optimizations—ping-pong batching and null dependency elimination—that improve its performance when there are 0 and 1 slow pilots respectively. Our evaluation of Copilot shows its performance is lower but competitive with Multi-Paxos and EPaxos when no replicas are slow. When a replica is slow, Copilot is the only protocol that avoids high latencies.

1 Introduction

Replicated state machines (RSMs) are linearizable, fault-tolerant groups of replicas coordinated by a consensus algorithm [46]. Linearizability gives the RSM the illusion of being a single machine that responds to client commands one by one [21]. Fault-tolerance enables the RSM to continue operating despite the failure of a minority of replicas. Together, these make RSMs operate as single machines that do not fail.

RSMs are used to implement small services that require strong consistency and fault tolerance, whose work can be handled by a single machine. They are used throughout large-scale systems, such as distributed databases [13, 14], cloud storage [6, 9], and service managers [25, 39]. While each RSM is individually small, their pervasive use at scale means that they collectively use many machines. At such scale, it is common for some machines to be slow [2, 15]. These slowdowns arise for a myriad of reasons, including misconfigurations, host-side network problems, partial hardware failures, garbage collection events, and many others. The slowdowns manifest as machines whose latency for responding to other machines is higher than usual.

Thus, RSMs should also be slowdown-tolerant, i.e., provide similar performance despite the presence of slow replicas. Unfortunately, no existing consensus protocol is slowdown-tolerant: a single slow replica can sharply increase their latency. This increased latency decreases availability because a service that does not respond in time is not meaningfully available [5, 6, 20, 48].

Slowdowns can be transient, lasting only a few seconds to minutes, or they can be long-term, lasting hours to days. Monitoring mechanisms within and around a system should eventually detect long-term slowdowns and reconfigure the slow replica out of the RSM to restore normal performance [1, 3, 23, 24, 32, 35]. What remains unsolved is how to tolerate transient slowdowns in general and how to tolerate long-term slowdowns in the time between their onset, their eventual detection, and the end of reconfiguration.

Our ultimate goal is to develop slowdown-tolerant RSMs that continue to operate as fast RSMs despite the presence of slow replicas. Given the general rarity of slowdowns, however, it is unlikely that a single RSM will contain multiple slow replicas at the same time. Thus, we target the first and most pragmatic step toward slowdown-tolerant RSMs: *1-slowdown-tolerant* RSMs that continue to provide normal performance despite the presence of 1 slow replica.

To provide 1-slowdown-tolerance, a consensus protocol must be able to tolerate a slowdown in all stages of processing a client's command: receive, order, execute, and reply. No existing consensus protocol is 1-slowdown-tolerant because none can handle a slow replica in the ordering stage. Existing ordering protocols all either rely on a single leader [3, 10, 28] or rely on the collaboration of multiple replicas [36, 40]. A single leader is not slowdown-tolerant because if it is slow, then it slows down the RSM. Multiple replicas collaboratively ordering commands is not slowdown-tolerant because if any of those replicas is slow, it slows down the RSM.

Copilot replication is the first 1-slowdown-tolerant consensus protocol. It avoids slowdowns using two distinguished replicas, the pilot and copilot. The two pilots do all stages of processing a client's command in parallel. This ensures all steps happen quickly even if one pilot is slow. Clients send commands to both pilots, and both pilots order, execute, and reply to the client. This proactive redundancy protects against a slowdown but also makes it more challenging to preserve consistency and efficiency.

The key challenge for Copilot replication is making its ordering stage slowdown tolerant. To provide linearizability, it needs to ensure the pilots agree on the ordering of client commands, but that in turn would naively require each to wait on the other if it is slow. Copilot instead allows a pilot to *fast takeover* the ordering work of a slow pilot. It does so by per-

sisting its takeover and subsequent ordering to the replicas.

Each pilot has a separate log where it orders client commands. Copilot combines the logs using dependencies, e.g., pilot log entry 9 is after copilot log entry 8. Copilot's ordering protocol has two phases—FastAccept, Accept—that commit commands to the pilots' logs along with their dependencies. In the FastAccept round, a pilot proposes an initial dependency for a log entry. If a sufficient number of replicas agree to this ordering, then this entry has committed on the *fast path* and the pilot moves on to execution. Otherwise—if the replicas have already agreed to a different ordering proposed by the other pilot—then the pilot adopts a dependency suggested by the replicas that it persists in the Accept round.

Copilot provides crash fault tolerance using similar mechanisms to Multi-Paxos [28, 37] that are applied independently to the log of each pilot. Copilot combines the logs of the two pilots using mechanisms inspired by EPaxos [40]. As such, it provides the same safety and liveness guarantees as Multi-Paxos and EPaxos. It is safe under any number of crash faults, and it is live as long as a majority of replicas can communicate in a timely manner. In addition, Copilot provides slowdown tolerance even if one replica is slow or failed.

The core Copilot protocol provides slowdown tolerance. However, it would naively go to the Accept round often as the two pilot's ordering commands continuously interleave and prevent one or both from taking the fast path. This additional round of messages would increase latency and decrease throughput relative to traditional consensus protocols like Multi-Paxos, which need only 1 round in the normal case.

Copilot replication includes two optimizations that keep it on the fast path almost all the time. When both pilots are fast, *ping-pong batching* coordinates them so that they alternate their proposals, allowing both pilots to commit on the fast path. When one pilot is slow, *null dependency elimination* allows the fast pilot to avoid waiting on commits from the slow pilot. With null dependency elimination, a fast pilot only needs to fast takeover the ordering work of the slow pilot that is in-progress when the slowdown begins.

Copilot replication is implemented in Go and our evaluation compares it to Multi-Paxos and EPaxos in a datacenter setting. When no replicas are slow, Copilot's performance is competitive with Multi-Paxos and EPaxos. When there is a slow replica, Copilot is the only consensus protocol that avoids high latencies for client requests.

In summary, this work makes the following contributions:

- Defining slowdown-tolerance and identifying why existing consensus protocols are not slowdown-tolerant (§2).
- Copilot replication, the first 1-slowdown-tolerant consensus protocol. Copilot replication uses two pilots to ensure the RSM stays fast, by using proactive redundancy in all stages of processing a client command (§3).
- Ping-pong batching and null dependency elimination, which make Copilot's performance with no slowdowns or one slowdown competitive with traditional protocols (§5).

2 Slowdown Tolerance

This section explains RSMs, defines slowdown tolerance, and explains why existing protocols do not tolerate slowdowns.

2.1 Replicated State Machine Primer

RSMs are linearizable, fault-tolerant groups of machines. They implement a state machine that atomically applies deterministic commands to stored state and returns any output [46]. The machines within an RSM are *replicas*. The RSM provides fault tolerance by starting the replicas in the same initial state and then moving them through the same sequence of states by executing commands in the same order. Then, if one of the replicas fails, the remaining replicas still have the state and can continue providing the service.

RSMs provide linearizability for client commands. *Linearizability* is a consistency model that ensures that client commands are (1) executed in some total order, and (2) this order is consistent with the real-time ordering of client commands, i.e., if command *a* completes in real-time before command *b* begins, then *a* must be ordered before *b* [21].

RSMs are coordinated by *consensus protocols* that determine a consistent order of client commands that are then applied across the replicas. An RSM goes through four stages to process a client command: it receives the command, it orders the command using the consensus protocol, it executes the command, and it replies to the client with any output. Each replica executes commands in the agreed-upon order. A common way to implement and think about RSMs is that they agree to put commands in sequentially increasing log entries, and then execute them in that log order.

2.2 Defining Slowdown Tolerance

We define a slow replica, clarify the relationship between slow and failed, and then define 1-slowdown-tolerance.

Defining a slow replica. We reason about the speed of a replica based on the time it takes between when the machine receives a request over the network and sends a response back out over the network. This includes the replica's RSM processing and its host-side network processing. It does not include the time it takes messages to traverse network links.

We say a replica is *slow* when its responses to messages take more than a threshold time t over its normal response time. For example, if a replica typically replies to messages within 1 ms, and we consider a slowdown threshold of $t = 10$ ms, then a replica is slow if it takes more than 11 ms to send responses. The precise setting of t will depend on the scenario and may even vary over time. For example, if an OS upgrade increases the processing speed of all replicas, then what was considered normal performance in the past may now be considered slow. We assume the term "slow" reflects the current definition and build our notion of slowdown tolerance on top of this term—that is, our notion of slowdown tolerance is robust to changes in what is considered slow.

Failed versus slow replicas. Replicas that have failed are also slow because they will not reply to messages within the slowdown threshold time. Thus, all failed replicas are slow. However, not all slow replicas are failed. Replicas can be slow but not failed for many reasons, e.g., misconfigurations, host-side network problems, or garbage collection events. It is these slow-but-not-failed replicas that we care about most because existing fault-tolerance mechanisms do not necessarily tolerate them.

Defining s-slowdown-tolerance. Traditionally, clients use RSMs because they provide a service that does not fail despite f replicas failing. Our definition of slowdown tolerance mirrors this traditional definition of fault tolerance while accounting for the dynamic nature of what is considered "slow." An RSM is *s-slowdown-tolerant* if it provides a service that is not slow despite s replicas being slow. More specifically, sort the replicas $\{r_1, ..., r_s, ..., r_n\}$ of an RSM according to the current definition of slow, such that $\{r_1, ..., r_s\}$ are the s slowest replicas. Let T represent how slow the RSM is—i.e., its response time properties based on the current definition of "slow"—and let T' represent how slow the RSM would be if replicas $\{r_1, ..., r_s\}$ were all replaced by clones of r_{s+1}. An RSM is s-slowdown-tolerant if the difference between T and T' is close to zero. In other words, the presence of s slow replicas should not appreciably slow down the RSM relative to an ideal scenario where those s replicas are not slow.

In this work, we focus on the practical case of 1-slowdown-tolerance. Designing RSMs that are s-slowdown-tolerant for $s > 1$ is an interesting avenue of future work.

2.3 Why Existing Protocols Slowdown

We explain why existing protocols are not slowdown tolerant using Multi-Paxos, EPaxos, and Aardvark as examples.

Multi-Paxos. Multi-Paxos [26, 28, 29, 37] is the canonical consensus protocol. It uses the replicas to elect a *leader*. The leader receives client commands and orders them by assigning them to the next available position in its log. It persists that order by sending Accept messages to the replicas and waiting for a majority quorum (including itself) to reply, which commits the command in that log position. It notifies other replicas of the commit using a Commit message. The replicas execute commands in the accepted prefix of the log in order, i.e., they only execute a command once its log position is committed and all previous log positions have been executed. After executing the command, the replicas reply to the client with any output. (We describe a variant of Multi-Paxos that has all replicas reply to the client, similar to PBFT [10], because it provides more redundancy.)

Figure 1a shows these steps and identifies parts of the protocol that are not slowdown tolerant. Receiving the client's command and running the ordering protocol are not slowdown tolerant because they are only done by the leader. If the leader is slow, it slows these stages. In turn, this is evident to clients whose commands see much higher latency.

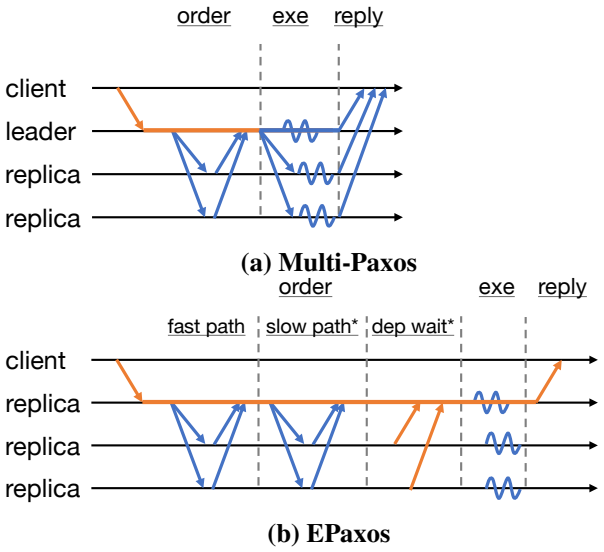


Figure 1: Message diagrams with execution for Multi-Paxos (a) and EPaxos (b). Orange components indicate parts of each protocol that are not slowdown tolerant because they lack redundancy. Blue components indicate parts with redundancy. EPaxos ordering phases that are only sometimes necessary are marked with asterisks (*).

Several parts of Multi-Paxos are individually slowdown tolerant—notably, the Accept messages sent to the replicas to persist the leader's ordering of a command. These messages are sent to all replicas with the leader only needing to hear back from a majority (including itself). For instance, with 5 replicas the leader sends the messages to the 4 other replicas and can proceed once it hears back from 2. This makes Multi-Paxos resilient to a non-leader replica being slow.

EPaxos. EPaxos [40] avoids the single leader of Multi-Paxos with a more egalitarian approach that distributes the work of receiving, ordering, executing, and replying across all replicas. Each replica in EPaxos receives commands from a subset of clients and runs the ordering protocol. We call this specific replica the command's *designated replica*. EPaxos's ordering protocol uses fine-grained dependencies between commands to dynamically determine an ordering using FastAccept and SlowAccept phases. Once a replica knows the dependencies of its commands, it waits for the final dependencies of its dependencies to arrive in the DependencyWait phase. Then a replica totally orders the commands and executes them in the resulting order. When a replica executes a command for which it is the designated replica, it sends the reply to the client. EPaxos can sometimes avoid the SlowAccept and DependencyWait phases.

Figure 1b shows these steps and identifies the parts of the protocol that are not slowdown tolerant. Receiving the client's command, running the ordering protocol, and replying to the client are all not slowdown tolerant because they are only done by a command's designated replica. If the des-

ignated replica is slow, it will slow down all of these stages, and thus the RSM, for its subset of clients.

DependencyWait can lead to slowdowns for all clients if any replica is slow. This is because DependencyWait requires a replica to wait until it learns the dependencies of the dependencies of a command. These transitive dependencies are necessary for EPaxos to consistently order commands at different replicas. But they are only determined and then sent from a command's designated replica. Thus, a slow replica will be slow to finalize and send out the dependencies for its designated commands to other replicas. This in turn slows commands that acquire dependencies on commands ordered by the slow replica, in addition to commands that use the slow replica as their designated replica.

Leader election. Consensus protocols with leaders include a leader election sub-protocol that provides fault tolerance in case a leader fails. In this sub-protocol, replicas detect when they think a leader may have failed, elect a new leader, ensure that the new leader's log includes all the commands that have been accepted by a majority quorum, and then have the new leader start processing new commands.

Some protocols, like Aardvark [3] and SDPaxos [51], have proposed using leader election to mitigate slowdowns as well, by having replicas detect when they think a leader is slow and then trigger the leader election sub-protocol. Unfortunately, this approach does not provide slowdown tolerance for two reasons. First, leader election is a heavy-weight process that makes an RSM unavailable while it is ongoing: no new commands can be processed until a new leader is elected and brought up to date. Second, leader election is only triggered when a replica thinks the leader is slow (or failed). Thus, only the subset of slowdowns detected by the replicas will be mitigated, and only after they have been detected. In contrast, 1-slowdown-tolerance requires an RSM to deliver performance as if the slowdown did not exist.

Consider the case of Aardvark. Aardvark employs two mechanisms to detect slowdowns in the leader: the first enforces a gradually increasing lower bound on the leader's throughput based on past peak performance; the second starts a heartbeat timer between each batch to ensure the leader is proposing new batches quickly enough. If the leader's throughput drops below the lower bound or if the heartbeat timeout expires, Aardvark initiates a view change to rotate the leader among the replicas. These mechanisms provide only partial slowdown tolerance because each limits the effects of only the subset of slowdowns it detects. For example, they do not protect against a replica whose processing path is slow for client requests but fast for replicas; or a replica whose responses become gradually slower over time while maintaining a small gap between successive responses. Such replicas would still be able to slow down the RSM during their turn as leader.

Further, using view changes to react to slowdowns can itself cause slowdowns and become costly. In practice, leader election timeouts are generally on the order of hundreds [43, 44] or thousands [8, 14, 17] of milliseconds to prevent the excess load, unavailability, and instability that occurs when leader elections are easily triggered. Thus, any leader slowdown whose severity is less than these timeouts will go undetected, as will any slowdown that is not covered by the detection mechanisms.

2.4 Summary and Insights

The fundamental problem with existing protocols is that they are *detection based*. Detection-based approaches do not protect against slowdowns until they are detected and never protect against slowdowns that are not detected. As a result, a consensus protocol cannot be 1-slowdown-tolerant if the path of a client's command includes at least one point where it goes through a single replica. If that replica is slow, the RSM will be slow (until and if the slowdown is detected). Thus, to design a 1-slowdown-tolerant replication protocol, we must *proactively* ensure there are at least 2 disjoint paths that a client's command can take at every stage. If one of these paths gets stuck at a slow replica, the other path can continue because we assume only 1 replica becomes slow.

3 Design

The core idea behind Copilot is to use two distinguished replicas, the pilot (P) and the copilot (P'), to redundantly process every client command. Figure 2 shows the life of an individual command in Copilot, which begins with a client sending the command to both pilots. By providing two disjoint paths for processing a command at every stage, Copilot prevents any single slow replica from slowing down the RSM.

This section describes the basic design of Copilot, and Section 5 describes optimizations that complete its design. This section first defines our model and then details each major part of the protocol—ordering, execution, and fast takeovers. Finally, it covers additional design details and summarizes why Copilot provides 1-slowdown-tolerance.

3.1 Model

Copilot assumes the *crash failure model*: a failed process stops executing and stops responding to messages. Copilot assumes an *asynchronous system*: there is no bound on the relative speed at which processes execute instructions, and there is no bound on the time it takes to deliver a message. Copilot requires $2f+1$ replicas to tolerate at most f failures, and guarantees linearizability as a correctness condition despite any number of failures. Copilot provides 1-slowdown-tolerance in the presence of any one slow replica.

3.2 Ordering

Copilot's ordering protocol places client commands into the *pilot log* and the *copilot log*, which are coordinated by the pilot and copilot, respectively. The two separate logs are ordered together using *dependencies* that indicate the prefix of

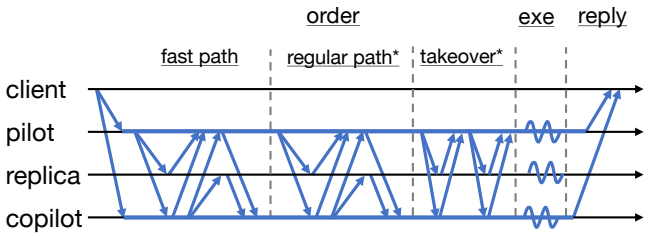


Figure 2: Message diagram with execution for Copilot. All components are in blue because all have the necessary redundancy to avoid any slow replica. Phases that are only sometimes necessary are marked with asterisks (*). The takeover phase only executes when it is necessary to prevent one pilot from waiting too long on the other pilot. Copilot's optimizations (§5) keep it on the fast path when both pilots are fast and mostly avoid the need for fast takeovers when one pilot is slow.

the other log that should be executed before a given entry. Pilots propose *initial dependencies* for log entries. Replicas either agree to that ordering or reply with a *suggested dependency*. Ultimately, each entry has a *final dependency* that is used by the execution protocol. The final dependencies between the pilot and copilot log may form cycles. Copilot's execution protocol constructs a single *combined log* using the final dependencies between the pilot and copilot logs and a priority rule that orders pilot entries in a cycle ahead of copilot entries. Figure 3 shows an example of how dependencies are used to order the entries in the combined log.

Copilot's ordering protocol persists the command and final dependency for a log entry to the replicas to ensure they can be recovered if up to f replicas (including both pilots) fail. The ordering protocol always includes a FastAccept phase and sometimes includes an Accept phase. The protocol completes after the FastAccept phase if enough of the replicas have agreed with the initial dependency to ensure it will always be recovered as the final dependency. Otherwise, the pilot selects a suggested dependency that orders an entry after enough of the other pilot's log to ensure linearizability.

The remainder of this subsection follows the ordering protocol in order, starting with the client sending a command to the replicas. Our description assumes no fast takeovers (§3.4) or view-changes (§3.5) for simplicity; with fast takeovers and view-changes, replicas reject messages when entries are taken over by another pilot, and entries can be committed with a no-op as a command.

Clients submit commands to both pilots. Each client has a unique client ID *cliid*. Clients assign commands a unique, increasing command ID *cid*. Clients send each command, its client ID, and its command ID to both pilots. The $\langle cliid, cid \rangle$ tuple uniquely identifies commands and enables the replicas to deduplicate them during execution.

Pilots propose commands and an initial dependency. Upon receiving a command from a client, a pilot puts the command into its next available log entry. It also assigns the *initial dependency* for this entry, which is the most recent entry from the other pilot it has seen. It then proposes this assignment of command and initial dependency for this entry to the other replicas by sending them FastAccept messages.

Replicas reply to FastAccepts. When a replica receives a FastAccept message it checks if the initial dependency for this entry is compatible with all previously accepted dependencies. If it is, the replica fast accepts the initial dependency. If it is not, the replica rejects the initial dependency and replies with a new suggested dependency.

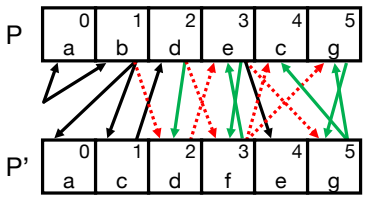
A pair of dependencies are *compatible* if at least one orders its entry after the other. Figure 3a shows examples of compatible and incompatible dependencies. $P'.1$ with dependency $P.1$, and $P.2$ with dependency $P'.1$ are compatible because $P.2$ is ordered after $P'.1$. $P'.3$ with dependency $P.2$ and $P.3$ with dependency $P'.2$ are *incompatible* because neither is ordered after the other. Incompatible dependencies must be avoided because they could lead to replicas with different subsets of the pilot and copilot logs executing entries in different orders, e.g., one replica executing $P.3$ then $P'.3$ and another executing $P'.3$ then $P.3$.

A replica uses the compatibility check to determine if an initial dependency, $P.i$ with dependency $P'.j$, is compatible with all previously accepted dependencies. $P.i$ is ordered after all previous entries in the P log automatically and after all entries $P'.j$ or earlier by its dependency. Thus, the check only needs to look at later entries in the other pilot's log. The *compatibility check* passes unless the replica has already accepted a later entry $P'.k$ ($k > j$) from the other pilot P' with a dependency earlier than $P.i$, i.e., $P'.k$'s dependency is $< P.i$.

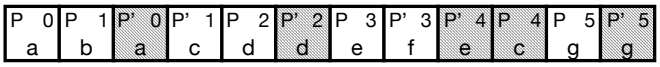
If it has not accepted a later entry, then this same check will prevent the replica from fast accepting any incompatible dependencies from the other pilot in the future. If it has accepted a later entry, but that entry's dependency is on $P.i$ or a later entry, then that entry, call it $P'.k$, is ordered after this one, i.e., $P'.j, P.i, \ldots, P'.k$. Thus, in either of these cases the replica fast accepts the initial dependency and replies with a FastAcceptOk message to the pilot. Otherwise, it sends a FastAcceptReply message to the pilot with its latest entry for the other pilot, $P'.k$, as its *suggested dependency*.

Pilots try to commit on the fast path. A pilot tries to gather a *fast quorum* of $f + \lfloor \frac{f+1}{2} \rfloor$ FastAcceptOk replies (including from itself).[1] If a pilot gathers a fast quorum, then enough replicas have agreed to its initial dependency that it will always be recovered from any majority quorum of replicas. Thus, it is safe for the pilot to commit this entry on the *fast path* and continue to execution. The entry's initial dependency is now its *final dependency* that is used during execution. The pilot also sends a Commit message to the other replicas to inform them of the final dependency for this entry. (It does not wait for responses for the Commit messages.)

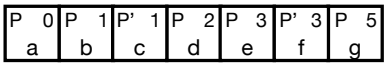
[1] This size is 2/3, 3/5, 5/7, and 6/9 for common RSM sizes.



(a) Dependencies join the two logs.



(b) Combined log with duplicates.



(c) Execution order from the combined log.

Figure 3: Dependencies are used to combine the pilot (P) and copilot (P') logs (a) into the combined log (b) that is deduplicated and then used for execution (c). (a) Solid black arrows indicate initial dependencies that became final dependencies because an entry was committed on the fast path. Dotted red arrows indicate initial dependencies rejected by the compatibility check because they could lead to different execution orders—e.g., $P.3$ or $P'.3$ could be executed seventh. Solid green arrows indicate final dependencies for entries whose initial dependency was rejected and thus committed on the regular path. Green arrows may contain cycles, which are consistently ordered by the execution protocol to derive a combined log. (b) The combined log has duplicates of most commands, shown in gray. (c) A command is only executed in its first position in the combined log.

A pilot might be unable to gather a fast quorum of FastAcceptOks for two reasons. First, it might receive FastAcceptReplys because replicas rejected the initial dependency as incompatible. Second, it might only receive as few as $f+1$ replies instead of the necessary f + $\lfloor\frac{f+1}{2}\rfloor$ because up to f of the $2f+1$ replicas have failed. In either case, the pilot waits until it receives at least $f+1$ FastAcceptOks and FastAcceptReplys and then continues to the Accept phase.

Pilots persist the final dependency in the Accept phase. A pilot selects the final dependency based on the suggested dependencies in the responses to the FastAccept round. All FastAcceptOk messages (including the pilot's) suggest the initial dependency. The pilot sorts the suggested dependencies in ascending order and then selects the $(f+1)$-th as the *final dependency*. This dependency is high enough to capture the necessary ordering constraint on this entry: it must use the $(f+1)$-th dependency to ensure quorum intersection with any command that has already been committed and potentially executed by the other pilot, so that this entry is ordered after that entry as required by linearizability. It is no higher to avoid creating more cycles for the other pilot: any dependency beyond the $(f+1)$-th will have its own dependency on this entry because this entry arrived at a majority quorum first.

Then the pilot persists this final dependency by sending it in an Accept message to all the other replicas. The ordering determined by final dependencies in Accept messages can create cycles at replicas. These cycles are acceptable because replicas will learn about them and then execute the commands in the cycles in the same order using the execution protocol. Thus, the other replicas accept this final dependency and reply with AcceptOk messages. When the pilot receives $f+1$ AcceptOks (including from itself) it has committed the entry on the *regular path*. It then sends Commit messages to the other replicas and proceeds to execution.

3.3 Execution

Replicas execute commands in the combined log order. The combined log contains each client command twice. A replica only executes a command in its first position in the combined order. After executing a command, the pilot and copilot reply to the command's client. Figure 3 shows an example of a combined log and its executed subset.

Copilot's total order of commands. The total order of commands in the combined log is determined by the partial order of each pilot's log, the dependencies between them, and a priority rule. There are three rules that define the total order. (1) The total order includes the partial order of each pilot's log, e.g., $P.0 < P.1 < P.2$ in Figure 3a. The dependencies between the logs sometimes create cycles. (2) When the dependencies are acyclic, the total order follows the dependency order, e.g., $P.1 < P'.0 < P'.1 < P.2$ in Figure 3a. (3) When the dependencies form a cycle, the total order is determined by the *priority* of the pilots: the pilot's entries are ordered before the copilot's, e.g., $P.4 < P.5 < P'.5$ in Figure 3a.

Executing in order. Replicas learn the final dependencies for each entry and thus use the same total order. A replica executes a command once its entry is committed and all preceding entries in the total order have been executed. The following rules determine when it is safe for a replica to execute a command in entry $P.i$ with dependency $P'.j$: (0) $P.i$ is committed, and (1) it has executed $P.(i-1)$, and then one of the following two conditions holds: (2) it has executed $P'.j$, or (3) $P.i$ and $P'.j$ are in a cycle and P is the pilot log. The rules 1–3 correspond to the rules that define the total order above.

Replicas can learn of committed entries out of order, e.g., a pilot can learn that their entries have committed before they learn of the commits for their dependencies. To ensure commands are executed in the total order, a replica must wait for the commit of all potentially preceding entries. For example, an entry in the pilot log $P.i$ must wait for the commit of all entries $< i$ in the pilot log, the commit of its dependency $P'.j$ in

the copilot log, and the commit of all entries $< j$ in the copilot log. Copilot's fast takeover protocol ensures a fast pilot need not wait long (§3.4) before executing this entry.

Deduplicating execution and replying. In the absence of failures, each command will be in the combined log twice. A replica executes each command only once in its first position. It tracks the commands from each client that have already been executed using the $\langle cliid, cid \rangle$ tuple. The first time it sees a command, it executes it. If the replica is the current pilot or copilot, it replies to the client with any output. The second time it sees a command, it simply marks it as executed and moves on. A client thus receives a response from each pilot for each command; it ignores the second response.

3.4 Fast Takeover

To execute commands in the total order determined by the ordering protocol, a pilot sometimes waits on commits from the other pilot. Waiting on the other pilot for a long time would not be slowdown tolerant. Copilot's fast takeover mechanism avoids a fast pilot waiting too long for a slow pilot by completing the necessary ordering work for that slow pilot.

All entries in the logs for both pilots have associated ballot numbers, and all messages include ballot numbers as in Paxos's proposal numbers [29]. These ballot numbers allow a fast pilot to safely takeover the work of a slow pilot using Paxos's two phases of prepare and accept. When a replica is elected as either pilot or copilot, that sets a ballot number for all entries in the corresponding log to be b. Replicas only (fast) accept entries if the included ballot number is $\geq$ the ballot number set for that entry. When a pilot is not slow, its included ballot numbers are exactly those set for each entry, and the protocol proceeds as described above.

When a pilot is slow, the other pilot can safely takeover its work by setting higher ballot numbers on the relevant entries in the slow pilot's log. The fast pilot does this by sending Prepare messages with a higher ballot number b' for the entry to all replicas. If b' is higher than the set ballot number for that entry, the replicas reply with PrepareOk messages and update their prepared ballot number for that entry. The PrepareOk messages indicate the progress of an entry at a replica, which is one of: not-accepted, fast-accepted, accepted, or committed. The PrepareOk messages include the highest ballot number for which a replica has fast or regular accepted an entry, the command and dependency associated with that entry, and an id of the dependency's proposing pilot.

After sending the Prepare messages, the fast pilot waits for at least $f+1$ PrepareOks (including from itself). If any of the PrepareOk messages indicate an entry is committed, the pilot short-circuits waiting and commits that entry with the same command and dependency. Otherwise, the fast pilot uses the value picking procedure described below to select a command and dependency. It then sends Accept messages for that command and final dependency, waits for $f+1$ AcceptOk replies, and then continues the execution protocol.

Recovery value picking procedure. We use *value* to indicate the command and dependency for a log entry. The fast takeover mechanism and view-change mechanism use the recovery value picking procedure to correctly recover a command and dependency for any entry that could have been committed and thus executed. This ensures all replicas execute all commands in the same combined log order.

The recovery value picking procedure is complex and its full details appear in our accompanying technical report [41]. The procedure examines the set S of PrepareOk replies that include the highest seen ballot number. The first three cases are straightforward:

1. There are one or more replies $r \in S$ with accepted as their progress. Then pick r's command and dependency.
2. There are $< \lfloor \frac{f+1}{2} \rfloor$ replies $r \in S$ with fast-accepted as their progress. Then pick no-op with an empty dependency.
3. There are $\geq f$ replies $r \in S$ with fast-accepted as their progress. Then pick r's command and dependency.

In the first case, the value may have been committed with a lower ballot number in an Accept phase, so the same value must be used. In the second case, the value could not have been committed in either an Accept phase or a FastAccept phase, so it is safe to pick a no-op. In the third case, the value may have been committed with a lower ballot number in a FastAccept phase and it is safe to use the same value. It is safe because the f or more fast-accept replies plus the entry's original proposing pilot form a majority quorum of replicas that passed the compatibility check. In turn, this ensures that any incompatible entries from the other pilot's log will be ordered after this entry. Thus, it is safe to commit this entry with its initial dependency.

The remaining case is when there are in the range of $[\lfloor \frac{f+1}{2} \rfloor, f)$ replies $r \in S$ with fast-accepted as their progress. In this case, the value may have been committed with a lower ballot number in a FastAccept phase, or it might not have because an incompatible entry in the other pilot's log reached the replicas first. In the first subcase we must commit using the same value, and in the second subcase we must not. To distinguish between these subcases, the recovering replica examines the first possible incompatible entry in the other pilot's log. If that entry is not yet committed, the recovering replica recovers that entry by repeating the above procedure, which enables it to safely distinguish between the subcases.

Triggering a fast takeover. A pilot sets a takeover-timeout when it has a committed command but does not know the final dependencies of all potentially preceding entries, i.e., it has not seen a commit for this entry's final dependency. If the takeover-timeout fires, the pilot stops waiting and does the necessary ordering work itself. It starts the fast takeover of all entries in the slow pilot's log that potentially precede this entry. Our implementation does this in a parallel batch for all entries. Setting the takeover-timeout too low could result in spurious fast takeovers that could lead to dueling proposers. We avoid dueling proposers using the standard

technique of randomized exponential backoff. We avoid spurious fast takeovers by setting a medium takeover-timeout in our implementation (10 ms). This medium timeout is fine because null dependency elimination (§5.2) avoids needing to wait when a pilot is continually slow.

Fast takeovers have a superficial resemblance to leader elections because both are triggered by one replica timing out while waiting to hear from another replica. Leader elections are triggered when one replica does not hear *something* from another replica—e.g., a heartbeat or a new proposal. But a leader can still send something regularly and/or quickly while being slow in other ways (§2.3). Fast takeovers, on the other hand, are triggered when one pilot is waiting to execute a specific client command. This puts them on the processing path of every request. When combined with the proactive redundancy of having both pilots process each client command, this bounds the latency of client commands to that of the faster pilot. If one pilot is slow, the other will process any given command up until execution and then, if necessary, wait for the takeover-timeout before completing the specific ordering work of the other pilot needed to unblock execution.

3.5 Additional Design

The additional parts of Copilot's design not described in this section all are similar to normal RSM designs. At-most-once semantics for client requests are handled using $\langle cliid,\ cid \rangle$ tuples and caching the output associated with a command. Non-deterministic commands can be handled by having pilots make the commands deterministic by doing the non-deterministic work (e.g., selecting a random number) and including it as input to the command. There will be two different non-deterministic versions of the command in the combined total order, but deduplication will ensure only the first is executed. State used for deduplication is garbage collected once a command is encountered in the log a second time.

Pilot and copilot election uses *view-changes*, analogous to Multi-Paxos's leader election [37], on the pilot and copilot logs, respectively. The view-change process has a newly elected pilot or copilot use the recovery value picking procedure described above while committing all unresolved entries in the log. The two separate logs of the pilots allow Copilot to elect a new pilot to replace a failed one while the other pilot continues to order and commit commands in its own log. While this is happening, the active pilot will acquire no new dependencies. Thus, the active pilot will be able to commit on the fast path and execute commands without waiting on any entries in the other log while a new pilot is elected.

3.6 Why Copilot is 1-Slowdown-Tolerant

Copilot achieves 1-slowdown tolerance by ensuring a client command is never blocked on a single path. That is, there are always two disjoint paths in the processing of a command, from when it is received by the RSM to when a response is sent to the client, and one of the paths must be fast.

When both pilots are fast, 1-slowdown tolerance is trivially achieved even if up to f (non-pilot) replicas are slow or failed. This is because the regular path only requires a majority of replicas, allowing both pilots' entries (and their dependencies) to commit and execute. If one of the pilots becomes slow or fails, then the other (fast) pilot can still commit its entries, but some of these entries might depend on uncommitted entries in the slow pilot's log. In this case, the fast pilot does a fast takeover of these entries and commits them. Thus, the fast pilot is able to continue executing its own entries. Shortly after a slowdown, the fast pilot stops acquiring dependencies on uncommitted entries (or acquires only null dependencies (§5.2)), eliminating the need for any fast takeovers. Thus, the performance of the RSM reduces to that of the faster pilot, satisfying 1-slowdown-tolerance.

4 Correctness

We prove that Copilot replication is both safe, i.e., it provides linearizability (4.1), and live, i.e., all client commands eventually complete (4.2). Our technical report [41] contains the full proofs; we summarize the intuition for each proof below.

4.1 Safety

To prove linearizability, we must show that client commands are (1) executed in some total order, and (2) this order is consistent with the real-time ordering of client operations, i.e., if command a completes in real-time before command b begins, then a must be ordered before b.

Let P and P' represent the two pilots. To prove the real-time ordering property, consider a command a that completes before a command b begins. Since a completes, it must be committed in at least one pilot's log; suppose w.l.o.g. it commits in P's log at entry $P.i$. Within P's log, a is trivially ordered before b, because b is issued only after a has been committed. In P''s log, a and b may commit in either order, but the key observation is that b's entry, call it $P'.j$, cannot have a dependency that precedes $P.i$, because this would be deemed incompatible during the FastAccept phase (cf. §3.2). Since $P'.j$'s dependency is $\geq P.i$ and $P.i$'s dependency is $< P'.j$, there are no cycles between $P.i$ and $P'.j$. Thus, $P.i$ is executed before $P'.j$, which implies that a is executed before b.

To prove the total ordering property, we first prove the following invariant: if two log entries $P.i$ and $P'.j$ commit at different pilots, either $P.i$ has a dependency $\geq P'.j$ or $P'.j$ has a dependency $\geq P.i$. This ensures that a dependency path exists from one entry to the other, preventing them from being ordered differently at different replicas. We then show that each entry in a pilot's log commits with the same commands and dependency across all replicas, even in the presence of failures (including failures of both pilots). This relies on the recovery value picking procedure from §3.4. When an entry commits on either the fast path or regular path, it is persisted to at least a majority of replicas. During a fast takeover or view change—which occur when one or both pilots are slow

(or failed)—the prepare phase will see the entry due to majority quorum intersection, and will reuse it when committing. If the replies from the prepare phase do not show a committed entry, then we must look at them more carefully. If any reply shows the entry is accepted, or if $\geq f$ replies show it is fast-accepted, then we commit the entry with its accepted dependency because it might have committed. If $< \lfloor \frac{f+1}{2} \rfloor$ replies show it is fast-accepted, then we can safely commit a no-op because the entry did not have enough fast accepts to commit. The final case occurs when the number of replies that show fast-accepted is in the range $[\lfloor \frac{f+1}{2} \rfloor, f)$. In this case, the entry may or may not have committed, depending on whether there was an incompatible entry in the other pilot's log. The recovery value picking procedure resolves this by examining and, if needed, recovering the first possible incompatible entry in the other pilot's log. Note that this procedure does not rely on replies from either pilot, and instead reasons about any $f+1$ possible replies received during the prepare phase.

Since each pilot's log is consistent across a majority of the replicas, the entries and their dependencies are also consistent, so the commands are executed in the same total order.

4.2 Liveness

To prove liveness, we must show that a command issued by a client eventually receives a response. Due to FLP [18], we assume the system is eventually partially-synchronous [16] and that all messages are eventually delivered.

Our proof uses a double induction. Assume a replica has executed all entries in P's log up to $P.i$ and all entries in P''s log up to $P'.k$. We show that the replica eventually executes either $P.(i+1)$ or $P'.(k+1)$, or a fast takeover occurs, or a view change occurs. Consider the failure-free case first.

If the dependency of $P.(i+1)$ is null or points to an entry $P'.j \leq P'.k$, then $P.(i+1)$ can be executed immediately. If $P'.j > P'.k$ (i.e., $P'.j$ has not been executed), then Copilot checks if a cycle exists between $P.(i+1)$ and $P'.j$. If no cycle exists, then execution switches to the next entry in P''s log, $P'.(k+1)$. $P'.(k+1)$ can be executed because its dependency must precede P_i (otherwise there would have been a cycle), which by our inductive assumption has been executed.

If there is a cycle and P has higher priority, Copilot breaks the cycle in favor of P and executes $P.(i+1)$. If P' has higher priority, execution switches to P''s log. Entry $P'.(k+1)$ can execute immediately if its dependency is $\leq P.i$ (by our inductive assumption), or after Copilot breaks the cycle in favor of P'. In all cases, either $P.(i+1)$ or $P'.(k+1)$ is executed.

Now consider the case of failures. If only non-pilots fail, this reduces to the failure-free case. If P' is slow/failed, then $P.(i+1)$ may not be able to execute because its dependency $P'.j$ may not have committed. In this case, P eventually does a fast takeover of $P'.j$'s entry. If both pilots are slow/failed, then neither $P.(i+1)$ nor $P'.(k+1)$ may be able to execute. In this case, a replica eventually initiates a view change to elect new pilots. Fast takeovers and view changes cannot repeat indefinitely by the same argument that basic Paxos and Multi-Paxos use to ensure progress, by relying on partial synchrony.

5 Optimizations

This section covers ping-pong batching and null dependency elimination, which improve Copilot's performance. Ping-pong batching coordinates the pilots so they propose compatible orderings when both are fast. Null dependency elimination allows a fast pilot to safely avoid waiting on commits from a slow pilot. Copilot includes both optimizations.

5.1 Ping-Pong Batching

Ping-pong batching coordinates the pilots so they propose compatible orderings to the replicas. The replicas fast accept these compatible orderings and thus the pilots commit on the fast path. With ping-pong batching, each pilot accumulates a batch of client commands. It assigns each command to its next available entry, so each batch is a growing assignment of client commands to consecutive entries. A pilot closes a batch and tries to FastAccept the batch when either it receives a FastAccept message from the other pilot or its ping-pong-wait timeout fires.

When both pilots are fast, they will close batches when they receive a FastAccept from the other pilot. This causes FastAccepts to ping-pong back and forth between the two pilots. The pilot closes its first batch and sends out its FastAccepts. When the copilot receives that FastAccept, it closes its first batch and sends out its FastAccepts. When the pilot receives that FastAccept, it closes its second batch, and so on.

This ping-ponging ensures that the pilots agree on the ordering of their entries. Before a pilot sends out a batch it hears about the latest batch from the copilot; and the copilot will not send out another batch until it hears about this batch from the pilot. Because the pilots agree on the ordering of their entries, the replicas can always fast accept their proposed orderings. If the replicas receive the proposed orderings in the same order that the pilots ping-pong propose them, then they agree to this ordering. Even when replicas receive the proposed ordering in a different order, they can still accept them because the dependencies will be compatible.

If one pilot is slow, the other will close its batches when the ping-pong-wait timeout fires. This timeout helps provide slowdown tolerance: even if one pilot is slow, the other need not wait on it for long.

5.2 Null Dependency Elimination

Null dependency elimination allows a fast pilot to avoid waiting on commits from a slow pilot. It looks inside a dependency to see the command it contains. If the contained command has already been executed, then execution deduplication (§3.3) will avoid executing it. We call these *null dependencies* because their execution will have no effect.

Sometimes a pilot must wait on the commit of the other pilot's earlier entries because it needs to know the finalized

dependency of that entry to know the agreed-upon total order. This is unnecessary for null dependencies because they are not executed. Thus, their final ordering information is irrelevant: a pilot need not determine when to execute them because it will not execute them. Instead, the pilot marks the null dependency as executed and continues.

When there is a continually slow pilot, null dependency elimination allows the fast pilot to avoid fast takeovers. A continually slow pilot will propose entries with a given command c after the fast pilot has already proposed an entry with that command c. Thus, the continually slow pilot's entries will be null dependencies for the fast pilot that can be safely skipped. This allows the fast pilot to never wait on commits from the slow pilot and thus avoids needing to fast takeover its entries. Fast takeovers are still necessary, however, for the cases when a pilot becomes slow *after* it proposes its ordering. Thus, when a pilot becomes slow, the other pilot does a fast takeover of the slow pilot's ongoing entries to provide 1-slowdown-tolerance. Thereafter, the fast pilot uses null dependency elimination to provide 1-slowdown-tolerance.

6 Evaluation

Copilot provides 1-slowdown-tolerant RSMs by using two pilots to provide redundancy at every stage of processing a command. Our evaluation demonstrates the benefit and quantifies the overhead of our approach. Specifically, it asks:

§6.3 Can Copilot tolerate transient slowdowns?
§6.4 Can Copilot tolerate slowdowns of varying severity?
§6.5 Can it tolerate slowdowns of varying manifestations?
§6.6 How does the throughput and latency of Copilot compare to existing consensus protocols?

Summary. We find that Copilot tolerates *any* one replica slowdown regardless of the type of slowdown, the role of the slow replica, or how slow the slow replica becomes. Copilot's latency under slowdown scenarios is comparable to its normal case latency when no replicas are slow. Copilot tolerates slowdowns better than Multi-Paxos, EPaxos, and Multi-Paxos with fast view changes. All commands in Multi-Paxos see high latencies when the leader is slow. EPaxos incurs a partial slowdown when any of the replicas is slow, and a slow replica can slow down other normal replicas under high conflict rates. Multi-Paxos with fast view changes tolerates the slowdowns that its low timeout detects, but it does not tolerate slowdowns that go undetected. Copilot achieves slowdown tolerance through redundancy. Althought this incurs more messages and processing, we find that Copilot's throughput and latency are competitive with Multi-Paxos and EPaxos.

6.1 Implementation and Baseline

We implemented Copilot in Go using the framework of EPaxos [40] to enable a fair comparison with the baselines. We use the framework's implementations of *EPaxos* and *Multi-Paxos*. The Multi-Paxos implementation is representative of well optimized Multi-Paxos [11, 26, 37]. Clients send commands directly to the leader, the leader gets those commands accepted in a single round of messages to the replicas, it executes the commands in log order, and then it replies to the clients. Replicas execute commands in log order but do not reply to the client. Any performance improvement we made to Copilot's implementation we also applied to EPaxos and Multi-Paxos to ensure the comparison remains fair.

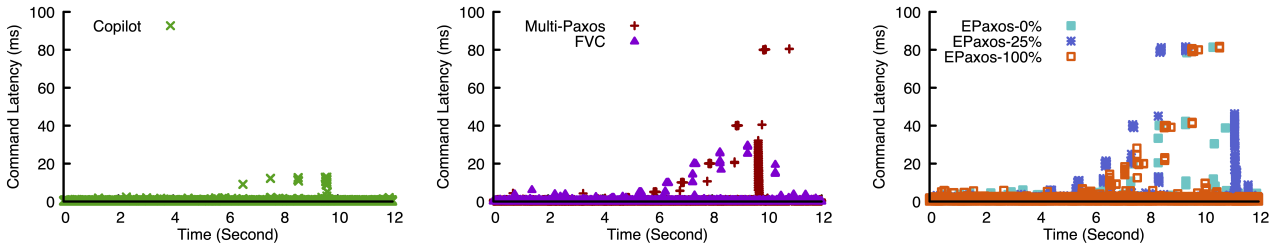
EPaxos and Multi-Paxos can use the *thrifty* optimization to send and receive messages only to the required number of other replicas. The thrifty optimization improves performance by decreasing load on all replicas in EPaxos and the leader in Multi-Paxos. It also harms slowdown-tolerance by eliminating redundancy from the ordering in these systems. Our latency slowdown experiments do not use the thrifty option for Multi-Paxos and EPaxos to show them in their best possible setting. Our throughput and latency experiments without slowdowns compare to the baselines with and without the thrifty optimization.

The EPaxos and Multi-Paxos baselines send pings every 3 s to make sure each replica has not failed. An alternative that would make them more slowdown tolerant, though less stable and unable to use some optimizations, is to use a very short view-change timeout. *Fast-View-Change* is a baseline we use to represent this alternative. Our implementation builds on the view-change implementation for Multi-Paxos in the EPaxos framework. It differs from a faithful implementation in two ways that decrease the time to complete a view change. Thus, its performance is an upper bound on that of a more faithful implementation. The first difference is that view-changes are triggered by a master process that never fails or becomes slow. The master receives heartbeats from the current leader every 1 ms and triggers a view-change as soon as 10 ms have elapsed with no heartbeats. (This timeout matches the fast-takeover timeout for Copilot.) The second difference is that a view-change immediately identifies the next leader instead of running an election, making the view-change process similar to that for viewstamped replication [34, 42]. If a client has not received a response to its command after 10 ms, it contacts the master to learn the current leader and resubmits its command to that leader.

6.2 Experimental Setup

Experiments were run on the Emulab testbed [49], where we have exclusive bare-metal access to 21 machines. Each machine has one 2.4 GHz 64-bit 8-Core processor, 64 GB RAM, and is networked with 1 Gbps Ethernet. These machines are located in the same datacenter with an average network round-trip time of about 0.1 ms. Thus, our evaluation of Copilot is focused on a datacenter setting with small latencies between replicas. Evaluation and optimization of Copilot for a geo-replicated setting is an interesting avenue of future work.

Configuration and workloads. We use 5 machines to create an RSM with 5 replicas that can tolerate at most 2 failures.



(a) Copilot. **(b) Multi-Paxos.** **(c) EPaxos.**

Figure 4: Client command latency for Copilot, Multi-Paxos, Fast-View-Change, and EPaxos with transient slowdowns. Transient slowdowns are injected every second starting at time 2 seconds. The severity and duration of the slowdowns in order are 0.5 ms, 1 ms, 2 ms, 5 ms, 10 ms, 20 ms, 40 ms, and 80 ms. Multi-Paxos and EPaxos have spikes in latency proportional to the slowdowns. Fast-View-Change tolerates the slowdowns using view changes to limit the maximum latency. Copilot tolerates the transient slowdowns because fast takeovers limit maximum latency.

We use 5-replica RSMs since they are a common setup for fault-tolerant services inside a datacenter [8]. Clients run on separate machines in the same facility. We use a simple workload with 8 byte commands that overwrite 4 bytes of data.

We run each experiment for 3 minutes and exclude the first and the last 30 seconds of each run to avoid experimental artifacts. To determine how to fairly configure our latency experiments, we probed the operation of each system under increasing load. For each system, we choose the number of closed-loop clients where the system operates at 50% of its peak load. This reduces the effect of queuing delays.

We enable batching for EPaxos and Multi-Paxos with a batching interval of 0.1 ms, which is similar to the effective length of Copilot's ping-pong batches. This choice of batching interval ensures all systems have similar median latency at low and moderate load. Copilot uses a ping-pong-wait timeout of 1 ms and a fast-takeover timeout of 10 ms.

For Multi-Paxos, clients send commands to the leader. For Copilot, clients send commands to both pilots. For EPaxos, each client has a designated replica it sends commands to.

EPaxos includes an interface that allows service builders to provide specialized logic in their implementation that identifies when two commands conflict. This allows EPaxos to avoid needing to determine an order between non-conflicting commands. We compare to EPaxos with 0%, 25%, and 100% conflicts. The 0% case is EPaxos's best case. The 100% case is EPaxos's worst case and also represents its performance when used as a generic RSM without its specialized interface. The 25% case is a middle ground.

Severity and duration. Slowdowns vary in their severity and their duration. The *severity* of a slowdown indicates its magnitude, e.g., a replica taking an extra 10 ms or an extra 80 ms to send responses. The *duration* of a slowdown indicates how long the slowdown lasts, e.g., 1 second or 10 minutes. For example, a replica could take an extra 10 ms to respond to every message it receives during a 1-second duration. We present experiments that evaluate tolerance of slowdowns of varying severity, duration, and manifestation.

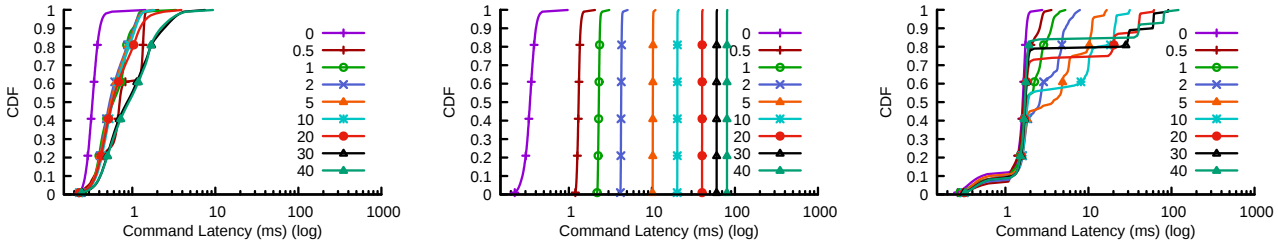
6.3 Transient Slowdowns

Figure 4 shows the latency of client commands for Copilot, Multi-Paxos, and EPaxos as transient slowdowns of increasing severity are injected. Transient slowdowns are injected every second starting at time 2 seconds. The injected slowdowns are pauses of increasing length, i.e., the severity and duration of the slowdown are both equal to the pause length. The pause lengths are 0.5 ms, 1 ms, 2 ms, 5 ms, 10 ms, 20 ms, 40 ms, and 80 ms. The pauses are injected by stopping all processing for the specified length inside the go processes. The slowdowns are injected on a pilot for Copilot, on the leader for Multi-Paxos, and on a replica for EPaxos.

Multi-Paxos and EPaxos slow down. Multi-Paxos and EPaxos each have latency spikes that increase proportionally with the length of the injected pause. For instance, for pauses of 40 ms, Multi-Paxos and EPaxos have commands with 40.1 ms and 41.5 ms respectively.

Fast-View-Change tolerates transient slowdowns. Fast-View-Change limits the maximum latency by detecting the pause and switching to a new leader. Maximum latency is controlled by the client timeout and view-change timeout. We see a maximum latency around their sum of 20 ms when a client needs to retransmit its command twice because the view-change had not completed after its first timeout. For instance, Fast-View-Change has commands with 25 ms latency for a 40 ms pause.

Copilot tolerates transient slowdowns. The latency for Copilot remains low and close to its latency when there are no slowdowns. For very small pauses, e.g., 0.5 ms, Copilot simply waits out the pause. This does not mask the slowdown and does show up in client command latency, but its magnitude is small enough that latency remains similar. For longer pauses, Copilot's fast-takeover timeout of 10 ms fires and the fast pilot completes the ordering work of the slow pilot. This keeps latency low and close to the timeout value. For instance, the maximum command latency is 12.6 ms for



(a) Copilot. **(b) Multi-Paxos.** **(c) EPaxos-25%.**

Figure 5: CDF of command latency for Copilot, Multi-Paxos, and EPaxos in the normal case (0) and with slowdowns of varying severity in ms. Slowdowns are injected for the duration of the experiment. Multi-Paxos and EPaxos have latency that increases proportionally with the severity of the slowdown. Copilot's latency stays low during the slowdowns because the fast pilot completes all stages of processing commands. In addition, null dependency elimination avoids having the fast pilot either wait on or fast takeover the ordering work of the slow pilot during the duration of a slowdown.

a 40 ms pause. The maximum latency during the onset of a slowdown is thus controlled by the fast-takeover timeout value. Latency as a slowdown continues, however, is even lower as our next experiment shows.

6.4 Slowdowns of Varying Severity

Figure 5 shows a CDF of latency for Copilot, Multi-Paxos, and EPaxos in the normal case (0 slowdown) and with slowdowns of varying severity that last for the duration of the experiment. A slowdown of the given severity is injected on one of the pilots for Copilot, the leader for Multi-Paxos, and a replica for EPaxos. The duration of these slowdowns is the length of the experiment (they last longer than the slowdowns evaluated in the previous subsection). The slowdowns are injected using Linux's traffic control (tc) to add delay corresponding to the severity on the slow replica. The severity ranges from 0.5 ms to 40 ms.

Multi-Paxos and EPaxos slow down. Figure 5b shows the CDF of latency for Multi-Paxos. The latency of client commands in Multi-Paxos is proportional to 2× the severity of the slowdown. The slowdown affects latency twice because the leader appears twice on the path for client commands: the message path is client-to-leader-to-replicas-to-leader-to-client. Fast-View-Change has similar results to Multi-Paxos when the severity of the slowdown is less than the view-change timeout and it avoids the slowdown using a view-change when the severity is greater than the timeout.

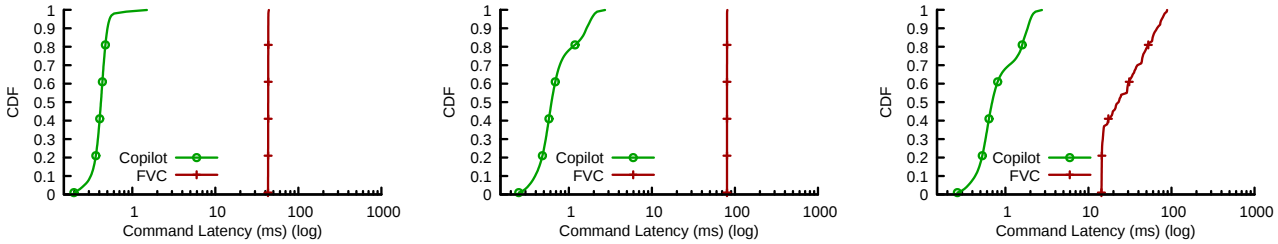
Figure 5c shows the CDF of latency for EPaxos with 25% conflicts. Normal case latency is higher than Multi-Paxos because EPaxos processes batches together, and if one command in a batch acquires a dependency then the entire batch goes to the slow path and does a dependency wait. With 25% conflicts, almost all batches have at least one command with a dependency and thus almost all have higher latency than Multi-Paxos. Slowdowns have two effects for EPaxos that result in two step functions in latency. First, the upper percentiles show a slowdown proportional to 2× the severity of the slowdown. This is due to the increased latency for commands whose designated replica is the slow replica. Second, the middle percentiles show a slowdown proportional to 1× the severity of the slowdown. This is due to the increased latency for commands that are ordered by a fast replica but that acquire a dependency on a command ordered by the slow replica. These commands wait on commits from the slow replica (§2.3). The CDF of latency for EPaxos with 0% conflicts (not shown) shows only the first effect. The CDF of latency for EPaxos with 100% conflicts (not shown) shows both effects with the latency of nearly all commands affected.

Copilot tolerates slowdowns of varying severity. Figure 5a show the CDF of latency for Copilot. Normal case latency is similar to Multi-Paxos. Copilot's latency under these slowdowns is related to its ping-pong-wait timeout of 1 ms. The fast pilot forms batches when either it hears from the slow pilot or its ping-pong-wait timeout fires. The fast pilot orders client commands in earlier batches than the slow pilot. Thus, null dependency elimination enables the fast pilot to avoid waiting on the slow pilot or having to fast takeover its work. The larger batches result in an increase in the latency for Copilot compared to its normal case, but this increase is small and overall performance is similar. Even in the worst case during a slowdown, median, 90th, and 99th percentile latencies are within 0.6 ms, 2 ms, and 4 ms of their values when there is no slowdown, respectively. Thus, we conclude that Copilot's implementation is resilient to slowdowns.

6.5 Slowdowns of Varying Manifestations

Figure 6 compares latency CDFs for Copilot and Fast-View-Change for three slowdowns with varying manifestations. The slowdowns are injected on the leader for Fast-View-Change and one of the pilots for Copilot.

Figure 6a considers a slowdown manifested by a slowed processing path for client commands with a fast processing path for messages from replicas. This experiment uses tc to inject 40 ms of delay. Fast-View-Change slows down in this case with 40 ms higher latency than usual because the client command processing path on the leader is slow.



(a) Slow for clients. (b) Slow with fast heartbeats. (c) Gradually slow.

Figure 6: CDF of client command latency for Copilot and Fast-View-Change with slowdowns of varying manifestations. Fast-View-Change's view changes are not triggered in these cases and latency spikes. Copilot's proactive redundancy tolerates these slowdowns and delivers latency similar to the normal case.

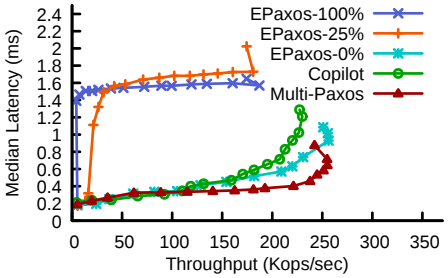


Figure 7: Throughput and latency without the thrifty optimization of the systems when there are no slow replicas.

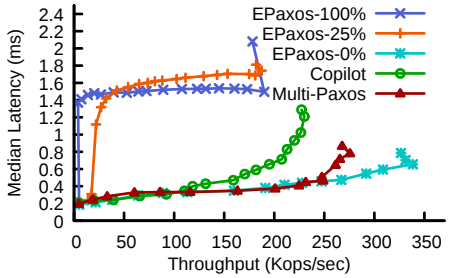


Figure 8: Throughput and latency with the thrifty optimization of the systems when there are no slow replicas.

Figure 6b shows a CDF of latency when the leader is slow but still quickly replies to heartbeats. This experiment injects 40 ms of delay to non-heartbeat processing directly in the Go process. Fast-View-Change slows down in this case with 80 ms higher latency than usual because the slow leader appears twice on the processing path for client commands.

Figure 6c shows a CDF of latency when the leader becomes gradually slower over time. The leader's processing of all messages (including heartbeats) is delayed by X ms, where X starts at 5 ms and increases by 1 ms every 1 second. This delay is directly injected in the Go process. Fast-View-Change slows down in this case with a CDF of latency that mirrors the increasing slowness of its leader.

In each of these slowdowns Fast-View-Change's low view change timeout is not triggered because the replicas are still regularly receiving messages from the leader. Multi-Paxos and EPaxos's view changes similarly would not be triggered. In contrast, Copilot's proactive redundancy tolerates these slowdowns and delivers latency similar to the normal case.

6.6 Performance Without Slow Replicas

Figure 7 shows the throughput and latency of the systems without the thrifty optimization as we increase load. We find that Copilot's throughput is about 8% lower than Multi-Paxos's. Copilot's latency at low/moderate load is similar to Multi-Paxos's; at high load its latency is higher but still low.

EPaxos's best case of 0% conflicts achieves the same peak throughput as Multi-Paxos with slightly higher latency. Under moderate and high conflict rates, EPaxos incurs another round-trip to commit on the slow path more often, and hence has higher latency and lower throughput. EPaxos processes an entire batch on the slow path if any command in the batch has a conflict. With 25% conflicts, almost all batches have at least one command with a conflict and thus almost all are processed on the slow path, resulting in similar performance to 100% conflicts. In contrast, Copilot and Multi-Paxos are not affected because they both totally order all commands.

Figure 8 shows the throughput and latency of all systems with the thrifty optimization as we increase load. Copilot does not use the thrifty optimization because its elimination of redundancy is not slowdown tolerant. Thus, Copilot's performance is the same. Multi-Paxos and EPaxos both see their maximum throughput increase. This makes EPaxos's best case (0% conflicts) provide clearly the highest throughput. With conflicts, however, its throughput is still lower than that of Copilot and Multi-Paxos. The thrifty optimization makes Multi-Paxos provide higher throughput than Copilot by about 35K commands/second, i.e., Copilot achieves 13% lower maximum throughput than Multi-Paxos. Multi-Paxos has higher throughput in this case because it needs to send and receive fewer messages.

Copilot's low latency and high throughput when there are no slow replicas is due to ping-pong batching. The pilots coordinate with each other to ensure that replicas agree with

their proposed ordering, allowing them to always commit on the fast path. Committing on the fast path keeps the amount of work each pilot needs to do for its own batches similar to that of a leader in Multi-Paxos. However, a pilot also needs to do the work of a replica for the other pilot's batches. Thus, Copilot's lower but competitive performance with Multi-Paxos is as we expect, because the pilots and leader are the throughput bottlenecks in each system respectively.

7 Related Work

This section reviews related work. To the best of our knowledge, all previous consensus protocols are not 1-slowdown-tolerant. Copilot's primary distinction is thus being the first 1-slowdown-tolerant consensus protocol. We review related work in consensus protocols, Byzantine consensus protocols, and slowdown cascades.

Consensus protocols. There is a growing body of consensus protocols that started with Paxos [28] and Viewstamped Replication [42]. New consensus protocols improve latency and/or throughput on these baselines [4, 22, 30, 31, 33, 36, 45, 51]. Others are designed to be more understandable [44]. SDPaxos [51] includes a throughput-based detection mechanism, similar to that of Aardvark (§2.3), that triggers a view-change for its sequencer that orders commands. Gryff unifies shared registers and consensus [7]. Its unproxied shared register operations are slowdown tolerant while its consensus operations are not. If the network ordering from NOPaxos [33] could be made slowdown tolerant, it could be used to eliminate the need for ping-pong batching to keep the pilots on the fast path in the normal case. To the best of our knowledge, none of these protocols are 1-slowdown-tolerant.

Paxos, EPaxos, Mencius. We drew inspiration in our design from Paxos, EPaxos, and Mencius. Our fast takeover protocol uses the classic 2-phase Paxos [28] on a slow pilot's log to enable a fast pilot to complete its ordering work. Our ordering protocol is influenced by EPaxos's ordering protocol [40]. It draws its use of dependencies and a multi-round ordering protocol with a fast path from EPaxos. Copilot's ordering differs because it orders the same commands twice, totally orders all commands, has only one dependency per entry, and includes fast takeovers. Mencius has all replicas work collaboratively to avoid doing redundant work or conflicting with each other [36]. Our ping-pong batching is inspired by Mencius and lets our pilots avoid conflicting with each other.

Byzantine consensus protocols. There is also a vast body of literature on Byzantine consensus protocols [3, 10, 12, 19, 27, 47, 50]. These protocols tolerate Byzantine faults, which Copilot does not. Most use the approach that PBFT introduced for practical systems of having multiple replicas execute a command and reply to the client. Copilot's use of both pilots to execute and reply to clients is inspired by this design.

Aardvark. Aardvark focuses on ensuring reliable minimum performance in BFT environments [3]. It employs two mechanisms to detect slowdowns in the leader: a gradually increasing lower bound on the leader's throughput, and an inter-batch heartbeat timer that ensures the leader is proposing new batches quickly enough. Both mechanisms trigger view changes to rotate the leader among replicas. As explained in §2.3, these mechanisms are detection based and hence provide only partial slowdown tolerance for Aardvark, because each limits the effect of a subset of slowdowns and incurs view changes that themselves cause slowdowns (§2.3). Copilot, in contrast, provides 1-slowdown-tolerance, because it *proactively* provides an alternative path for processing at all times, including during a view change to replace a slow pilot.

Note that Aardvark is designed for a Byzantine environment where replicas can be malicious. Copilot assumes nodes follow its protocol and thus would not work in a malicious setting. Focusing on crash faults allows Copilot to use techniques like fast takeovers and ping-pong batching to provide slowdown tolerance with good performance, which would be vulnerable to manipulation by a Byzantine replica. An interesting question to explore is whether mechanisms from Copilot and Aardvark can be combined to provide 1-slowdown-tolerance in a Byzantine environment.

Slowdown cascades. Occult is a scalable, geo-replicated data store that is immune to slowdown cascades [38]. Slowdown cascades occur when one slow shard of a scalable system cascades and affects other shards. They are a mostly orthogonal problem to slowdown tolerance because they are about preventing slowdowns of one part (shard) of a system from affecting other parts (shards) that do different work. Slowdown tolerance, in contrast, is about preventing slowdowns *within* an RSM, which may be one part (shard) of a larger system. Slowdown tolerance within shards decreases the likelihood of slowdown cascades. But they are mostly orthogonal, because cascades can still occur if there are more than s slowdowns within a shard.

8 Conclusion

Copilot replication is the first 1-slowdown-tolerant consensus protocol. Its pilot and copilot both receive, order, execute, and reply to all client commands. It uses this proactive redundancy and a fast takeover mechanism to provide slowdown tolerance. Despite its redundancy, Copilot replication's performance is competitive with existing consensus protocols when no replicas are slow. When a replica is slow, Copilot is the only consensus protocol that avoids high latencies.

Acknowledgements. We thank our shepherd, Allen Clement, and the anonymous reviewers for their insights and help in refining the ideas of this work. We are grateful to Christopher Hodsdon and Jeffrey Helt for their feedback. This work was supported by the National Science Foundation under grant number CNS-1827977.

References

[1] M. K. Aguilera and M. Walfish. No time for asynchrony. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[2] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[3] L. Alvisi, A. Clement, M. Dahlin, M. Marchetti, and E. Wong. Making byzantine fault tolerant systems tolerate byzantine faults. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[4] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up consensus by chasing fast decisions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[5] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.

[6] M. Brooker, T. Chen, and F. Ping. Millions of tiny databases. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[7] M. Burke, A. Cheng, and W. Lloyd. Gryff: Unifying consensus and shared registers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[8] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[9] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[11] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *ACM Symposium on Operating System Principles (SOSP)*, 2009.

[13] Cockroach DB. `https://www.cockroachlabs.com/product/`, 2020.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[15] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[17] etcd docs — Tuning. `https://etcd.io/docs/v3.4.0/tuning/`, 2020.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[19] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2010.

[20] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan. Meaningful availability. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[21] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1990.

[22] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2017.

[23] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles' heel of cloud-scale systems. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[24] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[25] M. Isard. Autopilot: automatic data center management. *Operating Systems Review*, 41(2):60–67, 2007.

[26] J. Kirsch and Y. Amir. Paxos for system builders: An overview. In *ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2008.

[27] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *ACM Symposium on Operating System Principles (SOSP)*, Oct. 2007.

[28] L. Lamport. The part-time parliament. *ACM Transac-*

tions on Computer Systems (TOCS), 16(2), 1998.

[29] L. Lamport. Paxos made simple. *ACM Sigact News*, 32, 2001.

[30] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[31] L. Lamport. Fast paxos. *Distributed Computing*, 19(2): 79–103, Oct. 2006.

[32] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[33] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[34] B. Liskov and J. Cowling. Viewstamped replication revisited. `http://www.pmg.lcs.mit.edu/papers/vr-revisited.pdf`, 2012.

[35] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[36] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec 2008.

[37] D. Mazières. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, 2007.

[38] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[39] Y. Mei, L. Cheng, V. Talwar, M. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, W. Chen, and G. J. Chen. Turbine: Facebook's Service Management Platform for Stream Processing. In *International Conference on Data Engineering (ICDE)*, 2020.

[40] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[41] K. Ngo, S. Sen, and W. Lloyd. Tolerating slowdowns in replicated state machines using copilots. Technical Report TR-004-20, Princeton University, Computer Science Department, 2020.

[42] B. M. Oki and B. H. Liskov. Viewstamped replication: A general primary copy. In *ACM Symposium on Principles of Distributed Computing (PODC)*, Aug. 1988.

[43] D. Ongaro. *Consensus: Bridging Theory And Practice*. PhD thesis, Stanford University, 2014.

[44] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[45] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[46] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computer Surveys*, 22(4), 1990.

[47] S. Sen, W. Lloyd, and M. J. Freedman. Prophecy: Using history for high-throughput fault tolerance. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[48] SLA summary for Azure services. `https://azure.microsoft.com/en-gb/support/legal/sla/summary/`, 2020.

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

[50] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. Zz and the art of practical BFT execution. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2011.

[51] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai. SDPaxos: Building efficient semi-decentralized geo-replicated state machines. In *ACM Symposium on Cloud Computing (SoCC)*, 2018.





Microsecond Consensus for Microsecond Applications

Marcos K. Aguilera
VMware Research

Naama Ben-David
VMware Research

Rachid Guerraoui
EPFL

Virendra J. Marathe
Oracle Labs

Athanasios Xygkis
EPFL

Igor Zablotchi
EPFL

Abstract

We consider the problem of making apps fault-tolerant through replication, when apps operate at the microsecond scale, as in finance, embedded computing, and microservices apps. These apps need a replication scheme that also operates at the microsecond scale, otherwise replication becomes a burden. We propose Mu, a system that takes less than 1.3 microseconds to replicate a (small) request in memory, and less than a millisecond to fail-over the system—this cuts the replication and fail-over latencies of the prior systems by at least 61% and 90%. Mu implements bona fide state machine replication/consensus (SMR) with strong consistency for a generic app, but it really shines on microsecond apps, where even the smallest overhead is significant. To provide this performance, Mu introduces a new SMR protocol that carefully leverages RDMA. Roughly, in Mu a leader replicates a request by simply writing it directly to the log of other replicas using RDMA, without any additional communication. Doing so, however, introduces the challenge of handling concurrent leaders, changing leaders, garbage collecting the logs, and more—challenges that we address in this paper through a judicious combination of RDMA permissions and distributed algorithmic design. We implemented Mu and used it to replicate several systems: a financial exchange app called Liquibook, Redis, Memcached, and HERD [33]. Our evaluation shows that Mu incurs a small replication latency, in some cases being the only viable replication system that incurs an acceptable overhead.

1 Introduction

Enabled by modern technologies such as RDMA, Microsecond-scale computing is emerging as a must [7]. A microsecond app might be expected to process a request in 10 microseconds. Areas where software systems care about microsecond performance include finance (e.g., trading systems), embedded computing (e.g., control systems), and microservices (e.g., key-value stores). Some of these areas are critical and it is desirable to replicate their microsecond apps across many hosts to provide high availability, due to economic, safety, or robustness reasons. Typically, a system may have hundreds of microservice apps [25], some of which are stateful and can disrupt a global execution if they fail (e.g., key-value stores)—these apps should be replicated for the sake of the whole system.

The golden standard to replicate an app is State Machine Replication (SMR) [68], whereby replicas execute requests in the same total order determined by a consensus protocol. Unfortunately, traditional SMR systems add hundreds of microseconds of overhead even on a fast network [28]. Recent work explores modern hardware in order to improve the performance of replication [30, 32, 36, 38, 61, 71]. The fastest of these (e.g., Hermes [36], DARE [61], and HovercRaft [38]) induce however an overhead of several microseconds, which is clearly high for apps that themselves take few microseconds. Furthermore, when a failure occurs, prior systems incur a prohibitively large fail-over time in the tens of *milli*seconds (not microseconds). For instance, HovercRaft takes 10 milliseconds, DARE 30 milliseconds, and Hermes at least 150 milliseconds. The rationale for such large latencies are timeouts that account for the natural fluctuations in the latency of modern networks. Improving replication and fail-over latencies requires fundamentally new techniques.

We propose Mu, a new SMR system that adds less than 1.3 microseconds to replicate a (small) app request, with the 99th-percentile at 1.6 microseconds. Although Mu is a general-purpose SMR scheme for a generic app, Mu really shines with microsecond apps, where even the smallest replication overhead is significant. Compared to the fastest prior system, Mu is able to cut 61% of its latency. This is the smallest latency possible with current RDMA hardware, as it corresponds to one round of *one-sided* communication.

To achieve this performance, Mu introduces a new SMR protocol that fundamentally changes how RDMA can be leveraged for replication. Our protocol reaches consensus and replicates a request with just one round of parallel RDMA write operations on a majority of replicas. This is in contrast to

prior approaches, which take multiple rounds [30,61,71] or resort to two-sided communication [28, 32, 39, 53]. Roughly, in Mu the leader replicates a request by simply using RDMA to write it to the log of each replica, without additional rounds of communication. Doing this correctly is challenging because concurrent leaders may try to write to the logs simultaneously. In fact, the hardest part of most replication protocols is the mechanism to protect against races of concurrent leaders (e.g., Paxos proposal numbers [40]). Traditional replication implements this mechanism using send-receive communication (two-sided operations) or multiple rounds of communication. Instead, Mu uses RDMA write permissions to guarantee that a replica's log can be written by only one leader. Critical to correctness are the mechanisms to change leaders and garbage collect logs, as we describe in the paper.

Mu also improves fail-over time to just 873 microseconds, with the 99-th percentile at 945 microseconds, which cuts fail-over time of prior systems by an order of magnitude. The fact that Mu significantly improves both replication overhead and fail-over latency is perhaps surprising: folklore suggests a trade-off between the latencies of replication in the fast path, and fail-over in the slow path.

The fail-over time of Mu has two parts: failure detection and leader change. For failure detection, traditional SMR systems typically use a timeout on heartbeat messages from the leader. Due to large variances in network latencies, timeout values are in the 10–100ms even with the fastest networks. This is clearly high for microsecond apps. Mu uses a conceptually different method based on a pull-score mechanism over RDMA. The leader increments a heartbeat counter in its local memory, while other replicas use RDMA to periodically read the counter and calculate a badness score. The score is the number of successive reads that returned the same value. Replicas declare a failure if the score is above a threshold, corresponding to a timeout. Different from the traditional heartbeats, this method can use an aggressively small timeout without false positives because network delays slow down the reads rather than the heartbeat. In this way, Mu detects failures usually within $\sim$600 microseconds. This is bottlenecked by variances in process scheduling, as we discuss later.

For leader change, the latency comes from the cost of changing RDMA write permissions, which with current NICs are hundreds of microseconds. This is higher than we expected: it is far slower than RDMA reads and writes, which go over the network. We attribute this delay to a lack of hardware optimization. RDMA has many methods to change permissions: (1) re-register memory regions, (2) change queue-pair access flags, or (3) close and reopen queue pairs. We carefully evaluate the speed of each method and propose a scheme that combines two of them using a fast-slow path to minimize latency. Despite our efforts, the best way to cut this latency further is to improve the NIC hardware.

We prove that Mu provides strong consistency in the form of linearizability [26], despite crashes and asynchrony, and it ensures liveness under the same assumptions as Paxos [40].

We implemented Mu and used it to replicate several apps: a financial exchange app called Liquibook [50], Redis, Memcached, and an RDMA-based key-value stored called HERD [33].

We evaluate Mu extensively, by studying its replication latency stand-alone or integrated into each of the above apps. We find that, for some of these apps (Liquibook, HERD), Mu is the only viable replication system that incurs a reasonable overhead. This is because Mu's latency is significantly lower by a factor of at least 2.7$\times$ compared to other replication systems. We also report on our study of Mu's fail-over latency, with a breakdown of its components, suggesting ways to improve the infrastructure to further reduce the latency.

Mu has some limitations. First, Mu relies on RDMA and so it is suitable only for networks with RDMA, such as local area networks, but not across the wide area. Second, Mu is an in-memory system that does not persist data in stable storage—doing so would add additional latency dependent on the device speed. [1] However, we observe that the industry is working on extensions of RDMA for persistent memory, whereby RDMA writes can be flushed at a remote persistent memory with minimum latency [70]—once available, this extension will provide persistence for Mu.

To summarize, we make the following contributions:

- We propose Mu, a new SMR system with low replication and fail-over latencies.
- To achieve its performance, Mu leverages RDMA permissions and a scoring mechanism over heartbeat counters.
- We give the complete correctness proof of Mu [2].
- We implement Mu, and evaluate both its raw performance and its performance in microsecond apps. Results show that Mu significantly reduces replication latencies to an acceptable level for microsecond apps.
- Mu's code is available at: https://github.com/LPD-EPFL/mu.

One might argue that Mu is ahead of its time, as most apps today are not yet microsecond apps. However, this situation is changing. We already have important microsecond apps in areas such as trading, and more will come as existing timing requirements become stricter and new systems emerge as the composition of a large number of microservices (§2.1).

[1] For fairness, all SMR systems that we compare against also operate in-memory.

2 Background

2.1 Microsecond Apps and Computing

Apps that are consumed by humans typically work at the millisecond scale: to the human brain, the lowest reported perceptible latency is 13 milliseconds [62]. Yet, we see the emergence of apps that are consumed not by humans but by other computing systems. An increasing number of such systems must operate at the microsecond scale, for competitive, physical, or composition reasons. Schneider [67] speaks of a microsecond market where traders spend massive resources to gain a microsecond advantage in their high-frequency trading. Industrial robots must orchestrate their motors with microsecond granularity for precise movements [6]. Modern distributed systems are composed of hundreds [25] of stateless and stateful microservices, such as key-value stores, web servers, load balancers, and ad services—each operating as an independent app whose latency requirements are gradually decreasing to the microsecond level [9], as the number of composed services is increasing. With this trend, we already see the emergence of key-value stores with microsecond latency (e.g., [32, 55]).

To operate at the microsecond scale, the computing ecosystem must be improved at many layers. This is happening gradually by various recent efforts. Barroso et al [7] argue for better support of microsecond-scale events. The latest Precision Time Protocol improves clock synchronization to achieve submicrosecond accuracy [4]. And other recent work improves CPU scheduling [9,58,63], thread management [65], power management [64], RPC handling [18, 32], and the network stack [58]—all at the microsecond scale. Mu fits in this context, by providing microsecond SMR.

2.2 State Machine Replication

State Machine Replication (SMR) replicates a service (e.g., a key-value storage system) across multiple physical servers called *replicas*, such that the system remains available and consistent even if some servers fail. SMR provides strong consistency in the form of linearizability [26]. A common way to implement SMR, which we adopt in this paper, is as follows: each replica has a copy of the service software and a log. The log stores client requests. We consider non-durable SMR systems [29, 31, 49, 52, 57, 59], which keep state in memory only, without logging updates to stable storage. Such systems make an item of data reliable by keeping copies of it in the memory of several nodes. Thus, the data remains recoverable as long as there are fewer simultaneous node failures than data copies [61].

A consensus protocol ensures that all replicas agree on what request is stored in each slot of the log. Replicas then apply the requests in the log (i.e., execute the corresponding operations), in log order. Assuming that the service is deterministic, this ensures all replicas remain in sync. We adopt a leader-based approach, in which a dynamically elected replica called the *leader* communicates with the clients and sends back responses after requests reach a majority of replicas. We assume a *crash-failure* model: servers may fail by crashing, after which they stop executing.

A consensus protocol must ensure *safety* and *liveness* properties. Safety here means (1) *agreement* (different replicas do not obtain different values for a given log slot) and (2) *validity* (replicas do not obtain spurious values). Liveness means *termination*—every live replica eventually obtains a value. We guarantee agreement and validity in an asynchronous system, while termination requires eventual synchrony and a majority of non-crashed replicas, as in typical consensus protocols. In theory, it is possible to design systems that terminate under weaker synchrony [14], but this is not our goal.

2.3 RDMA

Remote Direct Memory Access (RDMA) allows a host to access the memory of another host without involving the processor at the other host. RDMA enables low-latency communication by bypassing the OS kernel and by implementing several layers of the network stack in hardware.

RDMA supports many operations: Send/Receive, Write/Read, and Atomics (compare-and-swap, fetch-and-increment). Because of their lower latency, we use only RDMA Writes and Reads. RDMA has several transports; we use Reliable Connection (RC) to provide in-order reliable delivery.

RDMA connection endpoints are called Queue Pairs (QPs). Each QP is associated to a Completion Queue (CQ). Operations are posted to QPs as Work Requests (WRs). The RDMA hardware consumes the WR, performs the operation, and posts a Work Completion (WC) to the CQ. Applications make local memory available for remote access by registering local virtual memory regions (MRs) with the RDMA driver. Both QPs and MRs can have different access modes (e.g., read-only or read-write). The access mode is specified when initializing the QP or registering the MR, but can be changed later. MRs can overlap: the same memory can be registered multiple times, yielding multiple MRs, each with its own access mode. In this way, different remote machines can have different access rights to the same memory. The same effect can be obtained by using different access flags for the QPs used to communicate with remote machines.

3 Overview of Mu

3.1 Architecture

Figure 1 depicts the architecture of Mu. At the top, a client sends requests to an application and receives a response. We

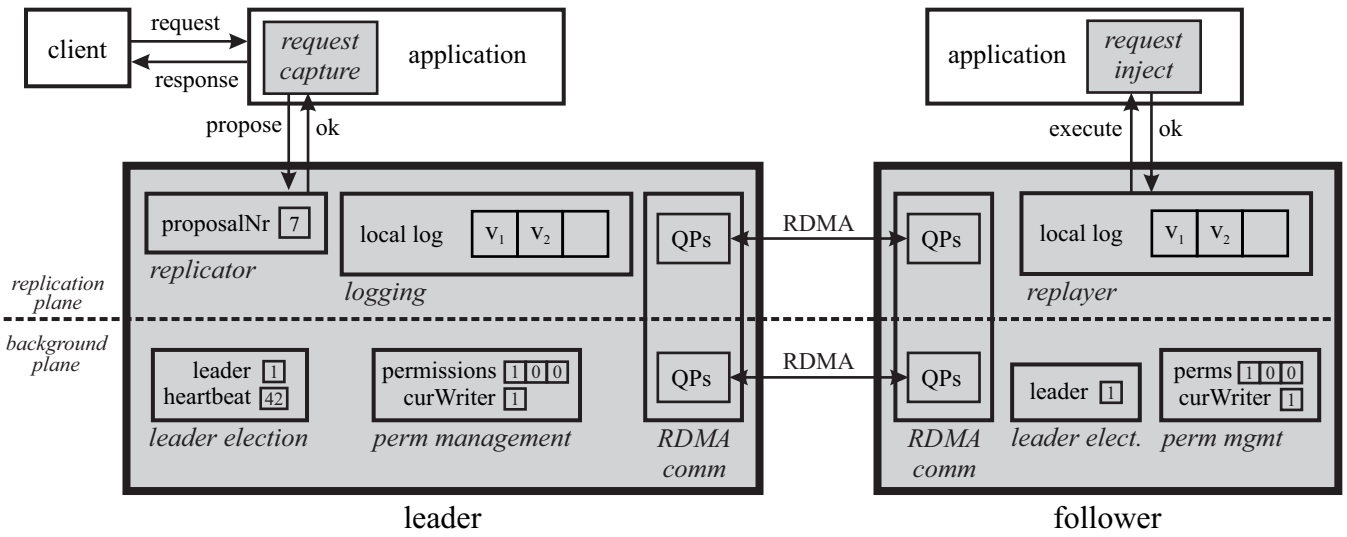


Figure 1: Architecture of Mu. Grey color shows Mu components. A replica is either a leader or a follower, with different behaviors. The leader captures client requests and writes them to the local logs of all replicas. Followers replay the log to inject the client requests into the application. A leader election component includes a heartbeat and the identity of the current leader. A permission management component allows a leader to request write permission to the local log while revoking the permission from other nodes.

are not particularly concerned about how the client communicates with the application: it can use a network, a local pipe, a function call, etc. We do assume however that this communication is amenable to being captured and injected. That is, there is a mechanism to capture requests from the client before they reach the application, so we can forward these requests to the replicas; a request is an opaque buffer that is not interpreted by Mu. Similarly, there is a mechanism to inject requests into the app. Providing such mechanisms requires changing the application; however, in our experience, the changes are small and non-intrusive. These mechanisms are standard in any SMR system.

Each replica has an idea of which replica is currently the leader. A replica that considers itself the leader assumes that role (left of figure); otherwise, it assumes the role of a follower (right of figure). Each replica grants RDMA *write permission* to its log for its current leader and no other replica. The replicas constantly monitor their current leader to check that it is still active. The replicas might not agree on who the current leader is. But in *the common case*, all replicas have the same leader, and that leader is active. When that happens, Mu is simple and efficient. The leader captures a client request, uses an RDMA Write to append that request to the log of each follower, and then continues the application to process the request. When the followers detect a new request in their log, they inject the request into the application, thereby updating the replicas.

The main challenge in the design of SMR protocols is to handle leader failures. Of particular concern is the case when a leader appears failed (due to intermittent network delays) so another leader takes over, but the original leader is still active.

To detect failures in Mu, the leader periodically increments a local counter: the followers periodically check the counter using an RDMA Read. If the followers do not detect an increment of the counter after a few tries, a new leader is elected.

The new leader revokes a write permission by any old leaders, thereby ensuring that old leaders cannot interfere with the operation of the new leader [3]. The new leader also reconstructs any partial work left by prior leaders.

Both the leader and the followers are internally divided into two major parts: the replication plane and the background plane. Roughly, the replication plane plays one of two mutually exclusive roles: the *leader role*, which is responsible for copying requests captured by the leader to the followers, or the *follower role*, which replays those requests to update the followers' replicas. The background plane monitors the health of the leader, determines and assigns the leader or follower role to the replication plane, and handles permission changes. Each plane has its own threads and queue pairs. This is in order to improve parallelism and provide isolation of performance and functionality. More specifically, the following components exist in each of the planes.

The replication plane has three components:

- *Replicator.* This component implements the main protocol to replicate a request from the leader to the followers, by writing the request in the followers' logs using RDMA Write.

- *Replayer.* This component replays entries from the lo-

cal log. This component and the replicator component are mutually exclusive; a replica only has one of these components active, depending on its role in the system.

- *Logging.* This component stores client requests to be replicated. Each replica has its own local log, which may be written remotely by other replicas according to previously granted permissions. Replicas also keep a copy of remote logs, which is used by a new leader to reconstruct partial log updates by older leaders.

The background plane has two components:

- *Leader election.* This component detects failures of leaders and selects other replicas to become leader. This is what determines the role a replica plays.

- *Permission management.* This component grants and revokes write access of local data by remote replicas. It maintains a permissions array, which stores access requests by remote replicas. Basically, a remote replica uses RDMA to store a 1 in this vector to request access.

We describe these planes in more detain in §4 and §5.

3.2 RDMA Communication

Each replica has two QPs for each remote replica: one QP for the replication plane and one for the background plane. The QPs for the replication plane share a completion queue, while the QPs for the background plane share another completion queue. The QPs operate in Reliable Connection (RC) mode.

Each replica also maintains two MRs, one for each plane. The MR of the replication plane contains the consensus log and the MR of the background plane contains metadata for leader election (§5.1) and permission management (§5.2). During execution, replicas may change the level of access to their log that they give to each remote replica; this is done by changing QP access flags. Note that all replicas always have remote read and write access permissions to the memory region in the background plane of each replica.

4 Replication Plane

The replication plane takes care of execution in the common case, but remains safe during leader changes. This is where we take care to optimize the latency of the common path. We do so by ensuring that, in the replication plane, only a leader replica communicates over the network, whereas all follower replicas are *silent* (i.e., only do local work).

In this section, we discuss algorithmic details related to replication in Mu. For pedagogical reasons, we first describe in §4.1 a basic version of the algorithm, which requires several round-trips to decide. Later, in §4.2, we discuss how Mu achieves its single round-trip complexity in the common case, as we present key extensions and optimizations to improve functionality and performance. We give an intuition of why the algorithm works in this section, and we provide the complete correctness argument in the full version of the paper [2].

4.1 Basic Algorithm

The leader captures client requests, and calls *propose* to replicate these requests. It is simplest to understand our replication algorithm relative to the Paxos algorithm, which we briefly summarize; for details, we refer the reader to [40]. In Paxos, for each slot of the log, a leader first executes a *prepare phase* where it sends a proposal number to all replicas.[2] A replica replies with either nack if it has seen a higher proposal number, or otherwise with the value with the highest proposal number that it has accepted. After getting a majority of replies, the leader adopts the value with the highest proposal number. If it got no values (only acks), it adopts its own proposal value. In the next phase, the *accept phase*, the leader sends its proposal number and adopted value to all replicas. A replica acks if it has not received any prepare phase message with a higher proposal number.

In Paxos, replicas actively reply to messages from the leader, but in our algorithm, replicas are silent and communicate information passively by publishing it to their memory. Specifically, along with their log, a replica publishes a *minProposal* representing the minimum proposal number which it can accept. The correctness of our algorithm hinges on the leader reading and updating the minProposal number of each follower before updating anything in its log, and on updates on a replica's log happening in slot-order.

However, this by itself is not enough; Paxos relies on active participation from the followers not only for the data itself, but also to avoid races. Simply publishing the relevant data on each replica is not enough, since two competing leaders could miss each other's updates. This can be avoided if each of the leaders reads the value after writing it [24]. However, this requires more communication. To avoid this, we shift the focus from the communication itself to the *prevention* of bad communication. A leader ℓ maintains a set of *confirmed followers*, which have granted write permission to ℓ and revoked write permission from other leaders before ℓ begins its operation. This is what prevents races among leaders in Mu. We describe these mechanisms in more detail below.

Log Structure

The main data structure used by the algorithm is the consensus log kept at each replica (Listing 1). The log consists of (1) a *minProposal* number, representing the smallest proposal number with which a leader may enter the accept phase on this

[2] Paxos uses proposer and acceptor terms; instead, we use leader and replica.

replica; (2) a *first undecided offset (FUO)*, representing the lowest log index which this replica believes to be undecided; and (3) a sequence of slots—each slot is a $(propNr, value)$ tuple.

Listing 1: Log Structure

```
1  struct Log {
2    minProposal = 0,
3    FUO = 0,
4    slots[] = (0,⊥) for all slots
5  }
```

Algorithm Description

Each leader begins its propose call by constructing its *confirmed followers* set (Listing 2, lines 9–12). This step is only necessary the first time a new leader invokes propose or immediately after an abort. This step is done by sending permission requests to all replicas and waiting for a majority of acks. When a replica acks, it means that this replica has granted write permission to this leader and revoked it from other replicas. The leader then adds this replica to its confirmed followers set. During execution, if the leader ℓ fails to write to one of its confirmed followers, because that follower crashed or gave write access to another leader, ℓ aborts and, if it still thinks it is the leader, it calls propose again.

After establishing its confirmed followers set, the leader invokes the prepare phase. To do so, the leader reads the *minProposal* from its confirmed followers (line 19) and chooses a proposal number *propNum* which is larger than any that it has read or used before. Then, the leader writes its proposal number into *minProposal* for each of its confirmed followers. Recall that if this write fails at any follower, the leader aborts. It is safe to overwrite a follower f's *minProposal* in line 22 because, if that write succeeds, then ℓ has not lost its write permission since adding f to its confirmed followers set, meaning no other leader wrote to f since then. To complete its prepare phase, the leader reads the relevant log slot of all of its confirmed followers and, as in Paxos, adopts either (a) the value with the highest proposal number, if it read any non-$\perp$ values, or (b) its own initial value, otherwise.

The leader ℓ then enters the accept phase, in which it tries to commit its previously adopted value. To do so, ℓ writes its adopted value to its confirmed followers. If these writes succeed, then ℓ has succeeded in replicating its value. No new value or minProposal number could have been written on any of the confirmed followers in this case, because that would have involved a loss of write permission for ℓ. Since the confirmed followers set constitutes a majority of the replicas, this means that ℓ's replicated value now appears in the same slot at a majority.

Finally, ℓ increments its own FUO to denote successfully replicating a value in this new slot. If the replicated value was ℓ's own proposed value, then it returns from the *propose* call; otherwise it continues with the prepare phase for the new FUO.

Listing 2: Basic Replication Algorithm of Mu

```
6  Propose(myValue):
7    done = false
8    If I just became leader or I just aborted:
9      For every process p in parallel:
10       Request permission from p
11       If p acks: add p to confirmedFollowers
12     Until this has been done for a majority
13   While not done:
14     Execute Prepare Phase
15     Execute Accept Phase

17 Prepare Phase:
18   For every process p in confirmedFollowers:
19     Read minProposal from p's log
20   Pick a new proposal number, propNum, higher
       ↪ than any minProposal seen so far
21   For every process p in confirmedFollowers:
22     Write propNum into LOG[p].minProposal
23     Read LOG[p].slots[myFUO]
24     Abort if any write fails
25   If all entries read were empty:
26     value = myValue
27   Else:
28     value = entry value with the largest
           ↪ proposal number of slots read

30 Accept Phase:
31   For every process p in confirmedFollowers:
32     Write propNum,value to p in slot myFUO
33     Abort if any write fails
34   If value == myValue:
35     done = true
36   Locally increment myFUO
```

4.2 Extensions

The basic algorithm described so far is clear and concise, but it also has downsides related to functionality and performance. We now address these downsides with some extensions, all of which are standard for Paxos-like algorithms; their correctness is discussed in the full version of our paper [2].

Bringing stragglers up to date. In the basic algorithm, if a replica r is not included in some leader's confirmed followers set, then its log will lag behind. If r later becomes leader, it can catch up by proposing new values at its current FUO, discovering previously accepted values, and re-committing them. This is correct but inefficient. Even worse, if r never becomes leader, then it will never recover the missing values. We address this problem by introducing an update phase for new leaders. After a replica becomes leader and establishes its confirmed followers set, but before attempting to replicate new values, the new leader (1) brings itself up to date with its highest-FUO confirmed follower (Listing 3) and (2) brings

its followers up to date (Listing 4). This is done by copying the contents of the more up-to-date log to the less up-to-date log.

Listing 3: Optimization: Leader Catch Up

```
1  For every process p in confirmedFollowers
2    Read p's FUO
3    Abort if any read fails
4  F = follower with max FUO
5  if F.FUO > myFUO:
6    Copy F.LOG[myFUO: F.FUO] into my log
7    myFUO = F.FUO
8    Abort if the read fails
```

Listing 4: Optimization: Update Followers

```
1  For every process p in confirmed followers:
2    Copy myLog[p.FUO: myFUO] into p.LOG
3    p.FUO = myFUO
4    Abort if any write fails
```

Followers commit in background. In the basic algorithm, followers do not know when a value is committed and thus cannot replay the requests in the application. This is easily fixed without additional communication. Since a leader will not start replicating in an index i before it knows index $i-1$ to be committed, followers can monitor their local logs and commit all values up to (but excluding) the highest non-empty log index. This is called *commit piggybacking*, since the commit message is folded into the next replicated value. As a result, followers replicate but do not commit the $(i-1)$-st entry until either the i-th entry is proposed by the current leader, or a new leader is elected and brings its followers up to date, whichever happens first.

Omitting the prepare phase. Once a leader finds only empty slots at a given index at all of its confirmed followers at line 23, then no higher index may contain an accepted value at any confirmed follower; thus, the leader may omit the prepare phase for higher indexes (until it aborts, after which the prepare phase becomes necessary again). This optimization concerns performance on the common path. With this optimization, the cost of a Propose call becomes a single RDMA write to a majority in the common case.

Growing confirmed followers. In the algorithm so far, the confirmed followers set remains fixed after the leader initially constructs it. This implies that processes outside the leader's confirmed followers set will miss updates, even if they are alive and timely, and that the leader will abort even if one of its followers crashes. To avoid this problem, we extend the algorithm to allow the leader to grow its confirmed followers set by briefly waiting for responses from all replicas during its initial request for permission. The leader can also add confirmed followers later, but must bring these replicas up to date (using the mechanism described above in *Bringing stragglers up to date*) before adding them to its set. When its confirmed follower set is large, the leader cannot wait for its RDMA reads and writes to complete at all of its confirmed followers before continuing, since we require the algorithm to continue operating despite the failure of a minority of the replicas; instead, the leader waits for just a majority of the replicas to complete.

Replayer. Followers continually monitor the log for new entries. This creates a challenge: how to ensure that the follower does not read an incomplete entry that has not yet been fully written by the leader. We adopt a standard approach: we add an extra *canary byte* at the end of each log entry [51, 71]. Before issuing an RDMA Write to replicate a log entry, the leader sets the entry's canary byte to a non-zero value. The follower first checks the canary and then the entry contents. In theory, it is possible that the canary gets written before the other contents under RDMA semantics. In practice, however, NICs provide left-to-right semantics in certain cases (e.g., the memory region is in the same NUMA domain as the NIC), which ensures that the canary is written last. This assumption is made by other RDMA systems [21, 22, 33, 51, 71]. Alternatively, we could store a checksum of the data in the canary, and the follower could read the canary and wait for the checksum to match the data.

5 Background Plane

The background plane has two main roles: electing and monitoring the leader, and handling permission change requests. In this section, we describe these mechanisms.

5.1 Leader Election

The *leader election component* of the background plane maintains an estimate of the current leader, which it continually updates. The replication plane uses this estimate to determine whether to execute as leader or follower.

Each replica independently and locally decides who it considers to be leader. We opt for a simple rule: replica i decides that j is leader if j is the replica with the lowest id, among those that i considers to be alive.

To know whether a replica has failed, we employ a *pull-score* mechanism, based on a *local heartbeat* counter. A leader election thread continually increments its own counter locally and uses RDMA Reads to read the counters (heartbeats) of other replicas and check whether they have been updated. It maintains a *score* for every other replica. If a replica has updated its counter since the last time it was read,

we increment that replica's score; otherwise, we decrement it. The score is capped by configurable minimum and maximum values, chosen experimentally to be 0 and 15, respectively. Once a replica's score drops below a *failure threshold*, we consider it to have failed; if its score goes above a *recovery threshold*, we consider it to be active and timely. To avoid oscillation, we have different *failure* and *recovery* thresholds, chosen experimentally to be 2 and 6, respectively, so as to avoid false positives.

Large network delays. Mu employs two timeouts: a small timeout in our detection algorithm (scoring), and a longer timeout built into the RDMA connection mechanism. The small timeout detects crashes quickly under common failures (process crashes, host crashes) without false positives. The longer RDMA timeout fires only under larger network delays (connection breaks, counter-read failures). In theory, the RDMA timeout could use exponential back-off to handle unknown delay bounds. In practice, however, that is not necessary, since we target datacenters with small delays.

Fate sharing. Because replication and leader election run in independent threads, the replication thread could fail or be delayed, while the leader election thread remains active and timely. This scenario is problematic if it occurs on a leader, as the leader cannot commit new entries, and no other leader can be elected. To address thie problem, every X=10000 iterations, the leader election thread checks the replication thread for activity; if the replication thread is stuck inside a call to `propose`, the replication thread stops incrementing the local counter, to allow a new leader to be elected.

5.2 Permission Management

The permission management module is used when changing leaders. Each replica maintains the invariant that only one replica at a time has write permission on its log. As explained in Section 4, when a leader changes in Mu, the new leader must request write permission from all the other replicas; this is done through a simple RDMA Write to a *permission request array* on the remote side. When a replica r sees a *permission request* from a would-be leader ℓ, r revokes write access from the current holder, grants write access to ℓ, and sends an ack to ℓ.

During the transition phase between leaders, it is possible that several replicas think themselves to be leader, and thus the permission request array may contain multiple entries. A permission management thread monitors and handles permission change requests one by one in order of requester id by spinning on the local permission request array.

RDMA provides multiple mechanisms to grant and revoke write access. The first mechanism is to register the consensus log as multiple, completely overlapping RDMA memory regions (MRs), one per remote replica. In order to grant or revoke access from replica r, it suffices to re-register the MR corresponding to r with different access flags. The second mechanism is to revoke r's write access by moving r's QP to a non-operational state (e.g., *init*); granting r write access is then done by moving r's QP back to the *ready-to-receive (RTR)* state. The third mechanism is to grant or revoke access from replica r by changing the access flags on r's QP.

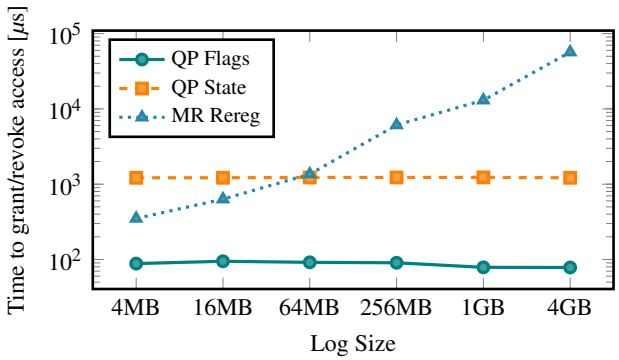


Figure 2: Performance comparison of different permission switching mechanisms. *QP Flags*: change the access flags on a QP; *QP Restart*: cycle a QP through the *reset*, *init*, *RTR* and *RTS* states; *MR Rereg*: re-register an RDMA MR with different access flags.

We compare the performance of these three mechanisms in Figure 2, as a function of the log size (which is the same as the RDMA MR size). We observe that the time to re-register an RDMA MR grows with the size of the MR, and can reach values close to 100ms for a log size of 4GB. On the other hand, the time to change a QPs access flags or cycle it through different states is independent of the MR size, with the former being roughly 10 times faster than the latter. However, changing a QPs access flags while RDMA operations to that QP are in flight sometimes causes the QP to go into an error state. Therefore, in Mu we use a fast-slow path approach: we first optimistically try to change permissions using the faster QP access flag method and, if that leads to an error, switch to the slower, but robust, QP state method.

5.3 Log Recycling

Conceptually, a log is an infinite data structure but in practice we need to implement a circular log with limited memory. This is done as follows. Each follower has a local *log head* variable, pointing to the first entry not yet executed in its copy of the application. The replayer thread advances the log head each time it executes an entry in the application. Periodically, the leader's background plane reads the log heads of all followers and computes *minHead*, the minimum of all log head pointers read from the followers. Log entries up to the minHead can be reused. Before these entries can be reused,

they must be zeroed out to ensure the correct function of the canary byte mechanism. Thus, the leader zeroes all follower logs after the leader's first undecided offset and before minHead, using an RDMA Write per follower. Note that this means that a new leader must first execute all leader change actions, ensuring that its first undecided offset is higher than all followers' first undecided offsets, before it can recycle entries. To facilitate the implementation, we ensure that the log is never completely full.

5.4 Adding and Removing Replicas

Mu adopts a standard method to add or remove replicas: use consensus itself to inform replicas about the change [40]. More precisely, there is a special log entry that indicates that replicas have been removed or added. Removing replicas is easy: once a replica sees it has been removed, it stops executing, while other replicas subsequently ignore any communication with it. Adding replicas is more complicated because it requires copying the state of an existing replica into the new one. To do that, Mu uses the standard approach of check-pointing state; we do so from one of the followers [71].

6 Implementation

Mu is implemented in 7157 lines of C++17 code (CLOC [19]). It uses the *ibverbs* library for RDMA over Infiniband. We implement all features and extensions in sections 4 and 5, except adding/removing replicas and fate sharing. Moreover, we implement some standard RDMA optimizations to reduce latency. RDMA Writes and Sends with payloads below a device-specific limit (256 bytes in our setup) are inlined: their payload is written directly to their work request. We pin threads to cores in the NUMA node of the NIC.

Our implementation is modular. We create several modules on top of the *ibverbs* library, which we expose as Conan [17] packages. Our modules deal with common practical problems in RDMA-based distributed computing (e.g., writing to all and waiting for a majority, gracefully handling broken RDMA connections etc.). Each abstraction is independently reusable. Our implementation also provides a QP exchange layer, making it straightforward to create, manage, and communicate QP information.

7 Evaluation

Our goal is to evaluate whether Mu indeed provides viable replication for microsecond computing. We aim to answer the following questions in our evaluation:

- What is the replication latency of Mu? How does it change with payload size and the application being replicated? How does Mu compare to other solutions?
- What is Mu's fail-over time?
- What is the throughput of Mu?

We evaluate Mu on a 4-node cluster, the details of which are given in Table 1. All experiments show 3-way replication, which accounts for most real deployments [28].

Table 1: Hardware details of machines.

CPU	2x Intel Xeon E5-2640 v4 @ 2.40GHz
Memory	2x 128GiB
NIC	Mellanox Connect-X 4
Links	100 Gbps Infiniband
Switch	Mellanox MSB7700 EDR 100 Gbps
OS	Ubuntu 18.04.4 LTS
Kernel	`4.15.0-72-generic`
RDMA Driver	Mellanox OFED `4.7-3.2.9.0`

We compare against APUS [71], DARE [61], and Hermes [36] where possible. The most recent system, HovercRaft [38], also provides SMR but its latency at 30–60 microseconds is substantially higher than the other systems, so we do not consider it further. For a fair comparison, we disable APUS's persistence to stable storage, since Mu, DARE, and Hermes all provide only in-memory replication.

We measure time using the POSIX `clock_gettime` function, with the `CLOCK_MONOTONIC` parameter. In our deployment, the resolution and overhead of `clock_gettime` is around 16–20*ns* [20]. In our figures, we show bars labeled with the median latency, with error bars showing 99-percentile and 1-percentile latencies. These statistics are computed over 1 million samples with a payload of 64-bytes each, unless otherwise stated.

Applications. We use Mu to replicate several microsecond apps: three key-value stores, as well as an order matching engine for a financial exchange.

The key-value stores that we replicate with Mu are Redis [66], Memcached [54], and HERD [33]. For the first two, the client is assumed to be on a different cluster, and connects to the servers over TCP. In contrast, HERD is a microsecond-scale RDMA-based key-value store. We replicate it over RDMA and use it as an example of a microsecond application. Integration with the three applications requires 183, 228, and 196 additional lines of code, respectively.

The other app is in the context of financial exchanges, in which parties unknown to each other submit buy and sell orders of stocks, commodities, derivatives, etc. At the heart of a financial exchange is an order matching engine [5], such as Liquibook [50], which is responsible for matching the buy and sell orders of the parties. We use Mu to replicate Liquibook. Liquibook's inputs are buy and sell orders. We created an unreplicated client-server version of Liquibook

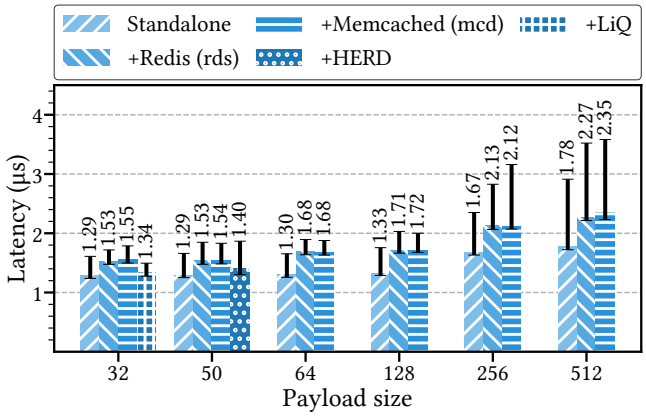


Figure 3: Replication latency of Mu integrated into different applications [Memcached (mcd), Liquibook (LiQ), Redis (rds), HERD] and payload sizes. Bar height and numerical labels show median latency; error bars show 99-percentile and 1-percentile latencies.

using eRPC [32], and then replicated this system using Mu. The eRPC integration and the replication required 611 lines of code in total.

7.1 Common-case Replication Latency

We begin by testing the overhead that Mu introduces in normal execution, when there is no leader failure. For these experiments, we first measure raw replication latency and compare Mu to other replication systems, as well as to itself under different payloads and attached applications.

Effect of Payload and Application on Latency We first study Mu in isolation, to understand its replication latency under different conditions.

We evaluate the raw replication latency of Mu in two settings: *standalone* and *attached*. In the standalone setting, Mu runs just the replication layer with no application and no client; the leader simply generates a random payload and invokes `propose()` in a tight loop. In the attached setting, Mu is integrated into one of a number of applications; the application client produces a payload and invokes `propose()` on the leader. These settings could impact latency differently, Mu and the application could interfere with each other.

Figure 3 compares standalone to attached runs as we vary payload size. Liquibook and Herd allow only one payload size (32 and 50 bytes), so they have only one bar each in the graph, while Redis and Memcached have many bars.

We see that the standalone version slightly outperforms the attached runs, for all tested applications and payload sizes. This is due to processor cache effects; in standalone runs, replication state, such as log and queue pairs, are always in cache, and the requests themselves need not be fetched from memory. This is not the case when attaching to an application. Additionally, in attached runs, the OS can migrate application threads (even if Mu's threads are pinned), leading to additional cache effects which can be detrimental to performance.

Mu supports two ways of attaching to an application, which have different processor cache sharing effects. The *direct* mode uses the same thread to run both the application and the replication, and so they share L1 and L2 caches. In contrast, the *handover* method places the application thread on a separate core from the replication thread, thus avoiding sharing L1 or L2 caches. Because the application must communicate the request to the replication thread, the handover method requires a cache coherence miss per replicated request. This method consistently adds ≈400ns over the standalone method. For applications with large requests, this overhead might be preferable to the one caused by the direct method, where replication and application compete for CPU time. For lighter weight applications, the direct method is preferable. In our experiments, we measure both methods and show the best method for each application: Liquibook and HERD use the direct method, while Redis and Memcached use the handover method.

We see that for payloads under 256 bytes, standalone latency remains constant despite increasing payload size. This is because we can RDMA-inline requests for these payload sizes, so the amount of work needed to send a request remains practically the same. At a payload of 256 bytes, the NIC must do a DMA itself to fetch the value to be sent, which incurs a gradual increase in overhead as the payload size increases. However, we see that Mu still performs well even at larger payloads quite well; at 512B, the median latency is only 35% higher than the latency of inlined payloads.

Comparing Mu to Other Replication Systems. We now study the replication time of Mu compared to other replication systems, for various applications. This comparison is not possible for every pair of replication system and application, because some replication systems are incompatible with certain applications. In particular, APUS works only with socket-based applications (Memcached and Redis). In DARE and Hermes, the replication protocol is bolted onto a key-value store, so we cannot attach it to the apps we consider—instead, we report their performance with their key-value stores.

Figure 4 shows the replication latencies of these systems. Mu's median latency outperforms all competitors by at least 2.7×, outperforming APUS on the same applications by 4×. Furthermore, Mu has smaller tail variation, with a difference of at most 500ns between the 1-percentile and 99-percentile latency. In contrast, Hermes and DARE both varied by more than 4μs across our experiments, with APUS exhibiting 99-percentile executions up to 20μs slower (cut off in the figure). We attribute this higher variance to two factors: the need to involve the CPU of many replicas in the critical path (Hermes

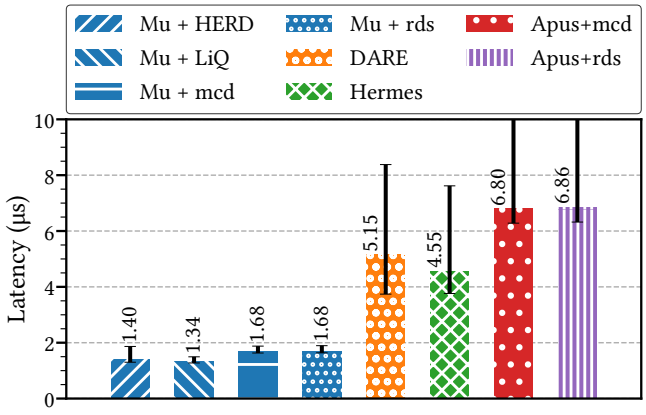


Figure 4: Replication latency of Mu compared with other replication solutions: DARE, Hermes, Apus on memcached (mcd), and Apus on Redis (rds). Bar height and numerical labels show median latency; error bars show 99-percentile and 1-percentile latencies.

and APUS), and sequentializing several RDMA operations so that their variance aggregates (DARE and APUS).

7.2 End-to-End Application Latency

Figure 5 shows the end-to-end latency of our tested applications, which includes the latency incurred by the application and by replication (if enabled). We show the result in three graphs corresponding to three classes of applications.

In all three graphs, we first focus on the unreplicated latency of these applications, so as to characterize the workload distribution. Subsequently, we show the latency of the same applications under replication with Mu and with competing systems, so as to exhibit the overhead of replication.

The leftmost graph is for Liquibook. The left bar is the unreplicated version, and the right bar is replicated with Mu. We can see that the median latency of Liquibook without replication is 4.08μs, and therefore the overhead of replication is around 35%. There is a large variance in latency, even in the unreplicated system. This variance comes from the client-server communication of Liquibook, which is based on eRPC. This variance changes little with replication. The other replication systems cannot replicate Liquibook (as noted before, DARE and Hermes are bolted onto their app, and APUS can replicate only socket-based applications). However, extrapolating their latency from Figure 4, they would add unacceptable overheads—over 100% overhead for the best alternative (Hermes).

The middle graph in Figure 5 shows the client-to-client latency of replicated and unreplicated microsecond-scale key-value stores. The first bars in orange show HERD unreplicated and HERD replicated with Mu. The green bar shows DARE's key-value store with its own replication system. The median unreplicated latency of HERD is 2.25μs, and Mu adds 1.34μs. While this is a significant overhead (59% of the original latency), this overhead is lower than any alternative. We do not show Hermes in this graph since Hermes does not allow for a separate client, and only generates its requests on the servers themselves. HERD replicated with Mu is the best option for a replicated key-value store, with overall median latency 2× lower than the next best option, and a much lower variance.

The rightmost graph in Figure 5 shows the replication of the traditional key-value stores, Memcached and Redis. The two leftmost bars show the client-to-client latencies of unreplicated Memcached and Redis, respectively. The four rightmost bars show the client-to-client latencies under replication with Mu and APUS. Note that the scale starts at 100μs to show better precision.

Mu incurs an overhead of around 1.5μs to replicate these apps, which is about 5μs faster than replicating with APUS. For these TCP/IP key-value stores, client-to-client latency under replication with Mu is around 5% lower than client-to-client latency under replication with APUS. With a faster client-to-app network, this difference would be bigger. In either case, Mu provides fault-tolerant replication with essentially no overhead for these applications.

Tail latency. From Figures 4 and 5, we see that applications replicated with DARE and APUS show large tail latencies and a skew towards lower values (the median latency is closer to the 1-st percentile than the 99-th percentile). We believe this tail latency occurs because DARE and APUS must handle several successive RDMA events on their critical path, where each event is susceptible to delay, thereby inflating the tail. Because Mu involves fewer RDMA events, its tail is smaller.

Figure 5 shows an even greater tail for the end-to-end latency of replicated applications. Liquibook has a large tail even in its unreplicated version, which we believe is due to its client-server communication, since the replication of Liquibook with Mu has a small tail (Figure 4). For Memcached and Redis, additional sources of tail latency are cache effects and thread migration, as discussed in Section 7.1. This effect is particularly pronounced when replicating with APUS (third panel of Figure 5), because the above contributors are compounded.

7.3 Fail-Over Time

We now study Mu's fail-over time. In these experiments, we run the system and subsequently introduce a leader failure. To get a thorough understanding of the fail-over time, we repeatedly introduce leader failures (1000 times) and plot a histogram of the fail-over times we observe. We also time the latency of permission switching, which corresponds to the time to change leaders after a failure is detected. The detection time is the difference between the total fail-over time and the permission switch time.

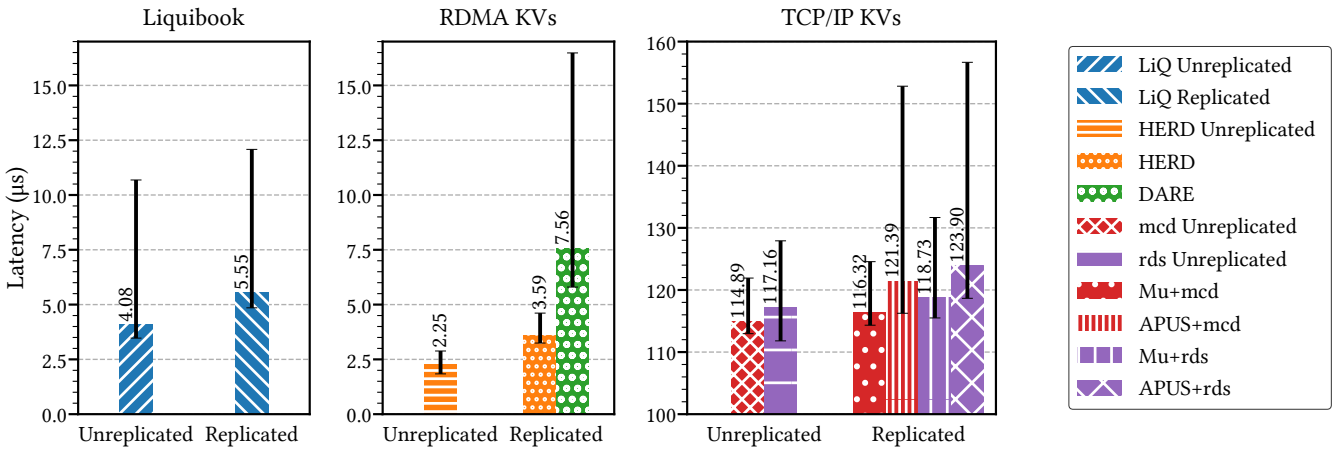


Figure 5: End-to-end latencies of applications. The first graph shows a financial exchange app (Liquibook) unreplicated and replicated with Mu. The second graph shows microsecond key-value stores: HERD unreplicated, HERD replicated with Mu, and DARE. The third graph shows traditional key-value stores: Memcached and Redis, unreplicated, as well as replicated with Mu and APUS. Bar height and numerical labels show median latency; error bars show 99-percentile and 1-percentile latencies.

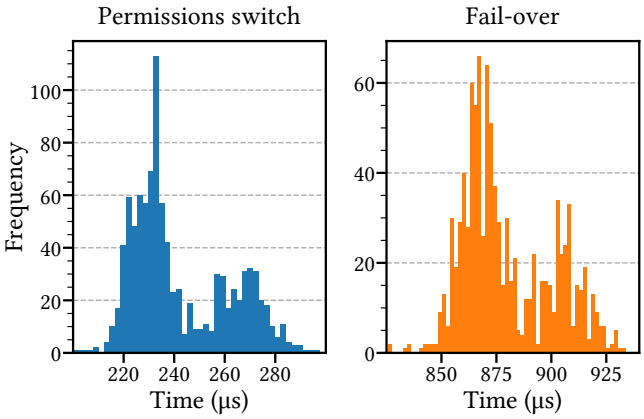


Figure 6: Fail-over time distribution.

We inject failures by delaying the leader, thus making it become temporarily unresponsive. This causes other replicas to observe that the leader's heartbeat has stopped changing, and thus detect a failure.

Figure 6 shows the results. We first note that the total fail-over time is quite low; the median fail-over time is 873μs and the 99-percentile fail-over time is 947μs, still below a millisecond. This represents an order of magnitude improvement over the best competitor at ≈10 ms (HovercRaft [38]).

The time to switch permissions constitutes about 30% of the total fail-over time, with mean latency at 244μs, and 99-percentile at 294μs. Recall that this measurement in fact encompasses two changes of permission at each replica; one to revoke write permission from the old leader and one to grant it to the new leader. Thus, improvements in the RDMA permission change protocol would be doubly amplified in Mu's fail-over time.

The rest of the fail-over time is attributed to failure detection (≈600μs). Although our pull-score mechanism does not rely on network variance, there is still variance introduced by process scheduling (e.g., in rare cases, the leader process is descheduled by the OS for tens of microseconds)—this is what prevented us from using smaller timeouts/scores and it is an area under active investigation for microsecond apps [9, 58, 63, 65].

7.4 Throughput

While Mu optimizes for low latency, in this section we evaluate the throughput of Mu. In our experiment, we run a standalone microbenchmark (not attached to an application). We increase throughput in two ways: by batching requests together before replicating, and by allowing multiple outstanding requests at a time. In each experiment, we vary the maximum number of outstanding requests allowed at a time, and the batch sizes.

Figure 7 shows the results in a latency-throughput graph. Each line represents a different max number of outstanding requests, and each data point represents a different batch size. As before, we use 64-byte requests.

We see that Mu reaches high throughput with this simple technique. At its highest point, the throughput reaches 47 Ops/μs with a batch size of 128 and 8 concurrent outstanding requests, with per-operation median latency at 17μs. Since the leader is sending requests to two other replicas, this translates to a throughput of 48Gbps, around half of the NIC bandwidth.

Latency and throughput both increase as the batch size increases. Median latency is also higher with more concurrent outstanding requests. However, the latency increases slowly, remaining at under 10μs even with a batch size of 64 and 8 outstanding requests.

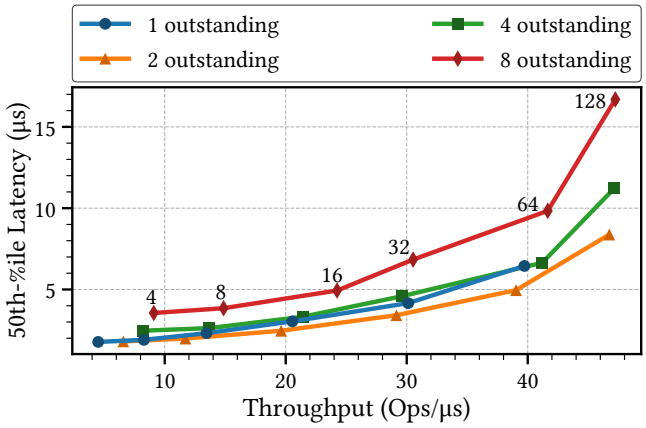


Figure 7: Latency vs throughput. Each line represents a different number of allowed concurrent outstanding requests. Each point on the lines represents a different batch size. Batch size shown as annotation close to each point.

There is a throughput wall at around 45 Ops/μs, with latency rising sharply. This can be traced to the transition between the client requests and the replication protocol at the leader replica. The leader must copy the request it receives into a memory region prepared for its RDMA write. This memory operation becomes a bottleneck. We could optimize throughput further by allowing direct contact between the client and the follower replicas. However, that may not be useful as the application itself might need some of the network bandwidth for its own operation, so the replication protocol should not saturate the network.

Increasing the number of outstanding requests while keeping the batch size constant substantially increases throughput at a small latency cost. The advantage of more outstanding requests is largest with two concurrent requests over one. Regardless of batch size, this allows substantially higher throughput at a negligible latency increase: allowing two outstanding requests instead of one increases latency by at most 400ns for up to a batch size of 32, and only 1.1μs at a batch size of 128, while increasing throughput by 20–50% depending on batch size. This effect grows less pronounced with higher numbers of outstanding requests.

Similarly, increasing batch size increases throughput with a low latency hit for small batch sizes, but the latency hit grows for larger batches. Notably, using 2 outstanding requests and a batch size of 32 keeps the median latency at only 3.4μs, but achieves throughput of nearly 30 Ops/μs.

8 Related Work

SMR in General. State machine replication is a common technique for building fault-tolerant, highly available services [40, 68]. Many practical SMR protocols have been designed, addressing simplicity [8,10,28,44,56], cost [39,43], and harsher failure assumptions [11, 12, 24, 39]. In the original scheme, which we follow, the order of all operations is agreed upon using consensus instances. At a high-level, our Mu protocol resembles the classical Paxos algorithm [40], but there are some important differences. In particular, we leverage RDMA's ability to grant and revoke access permissions to ensure that two leader replicas cannot both write a value without recognizing each other's presence. This allows us to optimize out participation from the follower replicas, leading to better performance. Furthermore, these dynamic permissions guide our unique leader changing mechanism.

Several implementations of Multi-Paxos avoid repeating Paxos's prepare phase for every consensus instance, as long as the same leader remains [13,41,53]. Piggybacking a commit message onto the next replicated request, as is done in Mu, is also used as a latency-hiding mechanism in [53,71].

Aguilera et al. [1] suggested the use of local heartbeats in a leader election algorithm designed for a theoretical message-and-memory model, in an approach similar to our pull-score mechanism. However, no system has so far implemented such local heartbeats for leader election in RDMA.

Single round-trip replication has been achieved in several previous works using two-sided sends and receives [23, 36, 37, 39, 43]. Theoretical work has shown that single-shot consensus can be achieved in a single one-sided round trip [3]. However, Mu is the first system to put that idea to work and implement one-sided single round trip SMR.

Alternative reliable replication schemes totally order only non-conflicting operations [16, 27, 36, 42, 59, 60, 69]. These schemes require opening the service being replicated to identify which operations commute. In contrast, we designed Mu assuming the replicated service is a black box. If desired, several parallel instances of Mu could be used to replicate concurrent operations that commute. This could be used to increase throughput in specific applications.

It is also important to notice that we consider "crash" failures. In particular, we assume nodes cannot behave in a Byzantine manner [11, 15, 39].

Improving the Stack Underlying SMR. While we propose a new SMR algorithm adapted to RDMA in order to optimize latency, other systems keep a classical algorithm but improve the underlying communication stack [32, 48]. With this approach, somewhat orthogonal to ours, the best reported replication latency is 5.5 μs [32], almost 5× slower than Mu. HovercRaft [38] shifts the SMR from the application layer to the transport layer to avoid IO and CPU bottlenecks on the leader replica. However, their request latency is more than an order of magnitude more than that of Mu, and they do not optimize fail-over time.

Some SMR systems leverage recent technologies such as programmable switches and NICs [29, 31, 49, 52]. However, programmable networks are not as widely available as RDMA, which has been commoditized with technologies

such as RoCE and iWARP.

Other RDMA Applications. More generally, RDMA has recently been the focus of many data center system designs, including key-value stores [21, 33] and transactions [35, 72]. Kalia et al. provide guidelines on the best ways to use RDMA to enhance performance [34]. Many of their suggested optimizations are employed by Mu. Kalia et al. also advocate the use of two-sided RDMA verbs (Sends/Receives) instead of RDMA Reads in situations in which a single RDMA Read might not suffice. However, this does not apply to Mu, since we know a priori which memory location should be read, and we rarely have to follow up with another read.

Failure detection. Failure detection is typically done using timeouts. Conventional wisdom is that timeouts must be large, in the seconds [47], though some systems report timeouts as low as 10 milliseconds [38]. It is possible to improve detection time using inside information [45, 47] or fine-grained reporting [46], which requires changes to apps and/or the infrastructure. This is orthogonal to our score-based mechanism and could be used to further improve Mu.

Similar RDMA-based Algorithms

A few SMR systems have recently been designed for RDMA [30, 61, 71], but used RDMA differently from Mu.

DARE [61] is the first RDMA-based SMR system. Similarly to Mu, DARE uses only one-sided RDMA verbs executed by the leader to replicate the log in normal execution, and makes use of permissions when changing leaders. However, unlike Mu, DARE requires updating the tail pointer of each replica's log in a separate RDMA Write from the one that copies over the new value, which leads to more round-trips for replication. DARE's use of permissions does not lead to a light-weight mechanism to block concurrent leaders, as in Mu. DARE has a heavier leader election protocol than Mu's, similar to that of RAFT, in which care is taken to ensure that at most one process considers itself leader at any point in time.

APUS [71] improves upon DARE's throughput. However, APUS requires active participation from the follower replicas during the replication protocol, resulting in higher latencies. Thus, it does not achieve the one-sided common-case communication of Mu. Similarly to DARE and Mu, APUS uses transitions through queue pair states to allow or deny RDMA access. However, like DARE, it does not use this mechanism to achieve a single one-sided communication round.

Derecho [30] provides durable and non-durable SMR, by combining a data movement protocol (SMC or RDMC) with a shared-state table primitive (SST) for determining when it is safe to deliver messages. This design yields high throughput but also high latency: a minimum of $10\mu s$ for non-durable SMR [30, Figure 12(b)] and more for durable SMR. This latency results from a node delaying the delivery of a message until all nodes have confirmed its receipt using the SST, which takes additional RDMA communication steps compared to Mu. It would be interesting to explore how Mu's protocol could improve Derecho.

Aguilera et al [3] present a one-shot consensus algorithm based on RDMA that solves consensus in a single one-sided communication round in the common case. They model RDMA's one-sided verbs as shared memory primitives which operate only if granted appropriate permissions. Their one-round communication complexity relies on changing permissions, an idea we use in Mu. While that work focuses on a theoretical construction, Mu is a fully fledged SMR system that needs many other mechanisms, such as logging, managing state, coordinating instances, recycling instances, handling clients, and permission management. Because these mechanisms are non-trivial, Mu requires its own proof of correctness [2]. Mu also provides an implementation and experimental evaluation not found in [3].

9 Conclusion

Computers have progressed from batch-processing systems that operate at the time scale of minutes, to progressively lower latencies in the seconds, then milliseconds, and now we are in the microsecond revolution. Work has already started in this space at various layers of the computing stack. Our contribution fits in this context, by providing generic microsecond replication for microsecond apps.

Mu is a state machine replication system that can replicate microsecond applications with little overhead. This involved two goals: achieving low latency on the common path, and minimizing fail-over time to maintain high availability. To reach these goals, Mu relies on (a) RDMA permissions to replicate a request with a single one-sided operation, as well as (b) a failure detection mechanism that does not incur false positives due to common network delays—a property that permits Mu to use aggressively small timeout values.

References

[1] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60, July 2018.

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor

Zablotchi. Microsecond consensus for microsecond applications. *ArXiv preprint arXiv:2010.06288*, October 2020. URL: https://arxiv.org/abs/2010.06288.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 409–418, July 2019.

[4] Anonymous. 1588-2019—IEEE approved draft standard for a precision clock synchronization protocol for networked measurement and control systems. https://standards.ieee.org/content/ieee-standards/en/standard/1588-2019.html.

[5] Order matching system. https://en.wikipedia.org/wiki/Order_matching_system.

[6] Anonymous. When microseconds count: Fast current loop innovation helps motors work smarter, not harder. http://e2e.ti.com/blogs_/b/thinkinnovate/archive/2017/11/14/when-microseconds-count-fast-current-loop-innovation-helps-motors-work-smarter-not-harder.

[7] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, April 2017.

[8] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *ACM SIGACT News*, 34(1):47–67, March 2003.

[9] Sol Boucher, Anuj Kalia, and David G. Andersen. Putting the "micro" back in microservice. In *USENIX Annual Technical Conference (ATC)*, pages 645–650, July 2018.

[10] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 335–350, November 2006.

[11] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 173–186, February 1999.

[12] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3), August 2003.

[13] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, August 2007.

[14] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.

[15] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, April 2009.

[16] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, November 2013.

[17] Conan, a C/C++ package manager. https://conan.io. Accessed 2020-09-30.

[18] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-driven tail-aware balancing of μs-scale RPCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 35–48, April 2019.

[19] Al Danial. cloc: Count lines of code. https://github.com/AlDanial/cloc.

[20] Travis Downs. A benchmark for low-level CPU micro-architectural features. https://github.com/travisdowns/uarch-bench.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, April 2014.

[22] Aleksandar Dragojevic, Dushyanth Narayanan, Ed Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[23] Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *International Conference on Dependable Systems and Networks (DSN)*, pages 22–27, June 2005.

[24] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed computing (DIST)*, 16(1):1–20, February 2003.

[25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna

Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, April 2019.

[26] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, January 1990.

[27] Brandon Holt, James Bornholt, Irene Zhang, Dan R. K. Ports, Mark Oskin, and Luis Ceze. Disciplined inconsistency with consistency types. In *Symposium on Cloud Computing (SoCC)*, pages 279–293, October 2016.

[28] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, June 2010.

[29] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 425–438, March 2016.

[30] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 36(2), April 2019.

[31] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, April 2018.

[32] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, February 2019.

[33] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM Conference on SIGCOMM*, pages 295–306, August 2014.

[34] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, pages 437–450, June 2016.

[35] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 185–201, November 2016.

[36] Antonios Katsarakis, Vasilis Avrielatos, M R Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojević, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 201–217, March 2020.

[37] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2):45–63, June 2001.

[38] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *European Conference on Computer Systems (EuroSys)*, April 2020.

[39] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, October 2007.

[40] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16:133–169, May 1998.

[41] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, June 2001.

[42] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[43] Leslie Lamport. Fast paxos. *Distributed computing (DIST)*, 19(2):79–103, July 2006.

[44] Butler W Lampson. How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms (WDAG)*, pages 1–17, October 1996.

[45] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[46] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed

systems with help from the network. In *European Conference on Computer Systems (EuroSys)*, April 2015.

[47] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the FALCON spy network. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[48] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: datacenter sockets can be fast and compatible. In *ACM Conference on SIGCOMM*, pages 90–103, August 2019.

[49] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 467–483, November 2016.

[50] Liquibook. https://github.com/enewhuis/liquibook. Accessed 2020-05-25.

[51] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, June 2004. URL: https://doi.org/10.1023/B:IJPP.0000029272.69895.c1, doi:10.1023/B:IJPP.0000029272.69895.c1.

[52] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *ACM Conference on SIGCOMM*, pages 318–333, August 2019.

[53] David Mazieres. Paxos made practical. https://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, 2007.

[54] Memcached. https://memcached.org/. Accessed 2020-05-25.

[55] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *ACM International Conference on Systems and Storage (SYSTOR)*, pages 97–108, May 2019.

[56] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, pages 305–319, June 2014.

[57] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41, October 2011.

[58] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, February 2019.

[59] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 47–64, February 2019.

[60] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, April 2002.

[61] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on RDMA networks. In *Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 107–118. ACM, June 2015.

[62] Mary C. Potter, Brad Wyble, Carl Erick Hagmann, and Emily Sarah McCourt. Detecting meaning in RSVP at 13 ms per picture. *Attention, Perception, & Psychophysics*, 76(2):270–279, February 2014.

[63] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, October 2017.

[64] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Symposium on Cloud Computing (SoCC)*, pages 342–355, August 2015.

[65] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 145–160, October 2018.

[66] Redis. https://redis.io/. Accessed 2020-05-25.

[67] David Schneider. The microsecond market. *IEEE Spectrum*, 49(6), June 2012.

[68] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM*

Computing Surveys (CSUR), 22(4):299–319, December 1990.

[69] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400, October 2011.

[70] SNIA. Extending RDMA for persistent memory over fabrics. https://www.snia.org/sites/default/files/ESF/Extending-RDMA-for-Persistent-Memory-over-Fabrics-Final.pdf.

[71] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and scalable paxos on RDMA. In *Symposium on Cloud Computing (SoCC)*, pages 94–107, September 2017.

[72] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, October 2015.

Virtual Consensus in Delos

Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi
Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski
Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, Yee Jiun Song
Facebook, Inc.

Abstract

Consensus-based replicated systems are complex, monolithic, and difficult to upgrade once deployed. As a result, deployed systems do not benefit from innovative research, and new consensus protocols rarely reach production. We propose virtualizing consensus by virtualizing the shared log API, allowing services to change consensus protocols without downtime. Virtualization splits the logic of consensus into the VirtualLog, a generic and reusable reconfiguration layer; and pluggable ordering protocols called Loglets. Loglets are simple, since they do not need to support reconfiguration or leader election; diverse, consisting of different protocols, codebases, and even deployment modes; and composable, via RAID-like stacking and striping. We describe a production database called Delos[1] which leverages virtual consensus for rapid, incremental development and deployment. Delos reached production within 8 months, and 4 months later upgraded its consensus protocol without downtime for a 10X latency improvement. Delos can dynamically change its performance properties by changing consensus protocols: we can scale throughput by up to 10X by switching to a disaggregated Loglet, and double the failure threshold of an instance without sacrificing throughput via a striped Loglet.

1 Introduction

The last decade has seen significant research advances in faster and more flexible consensus protocols. Unfortunately, systems that use consensus to replicate state are monolithic, complex, and difficult to evolve. As a result, deployed systems rarely benefit from new research ideas (e.g., ZooKeeper [20] still runs a decade-old protocol [22]); in turn, such ideas only have real-world impact when entire new production systems and applications are built from scratch around them (e.g., VMware's CorfuDB [1] uses sharded acceptors [7, 16]; Facebook's LogDevice [3] implements flexible quorums [19]; etcd [2] runs on Raft [39]). Contrast this state of affairs with other areas such as OSes and networks, where modular design and clean layering allow plug-and-play adoption of new mechanisms and incremental improvement of existing ones: for example, a new type of SSD, a new filesystem layout, or a new key-value store like RocksDB can each be deployed with no modification to the layers above or below it.

Recently, the shared log has gained traction as an API for consensus in research [7–9, 16, 37] and industry [1, 3, 23, 47]. Applications can replicate state via this API by appending updates to the shared log, checking its tail, and reading back updates from it. The consensus protocol is hidden behind the shared log API, allowing applications to bind to any implementation at deployment time.

Unfortunately, an API on its own is not sufficient to enable incremental evolution. First, new implementations of the shared log are *difficult to deploy and operate*: no support exists for upgrading and migrating applications to different implementations without downtime, which is untenable for highly available services. Second, new implementations are *difficult to develop*: the consensus protocol implementing the shared log is itself a complex distributed system containing a data plane (for ordering and storing commands durably) and a control plane (for reconfiguring leadership, roles, parameters, and membership). Existing protocols such as Raft aggressively combine both planes into a single protocol; in doing so, they give up the ability to incrementally change the data plane (i.e., the ordering mechanism) without reimplementing the entire control plane. As a result of these two limitations, systems have to be written and deployed from scratch around new consensus protocols (e.g., ZooKeeper cannot be upgraded to run over Raft [39] or CORFU [7]); and protocols have to be rewritten around new ordering mechanisms (e.g., Raft cannot be changed easily to support sharded acceptors).

In this paper, we virtualize consensus by virtualizing the shared log API. We propose the novel abstraction of a virtualized shared log (or *VirtualLog*). The VirtualLog exposes a conventional shared log API; applications above it are oblivious to its virtualized nature. Under the hood, the VirtualLog

[1] Delos is an island in the Cyclades, a few hundred miles from Paxos and Corfu.

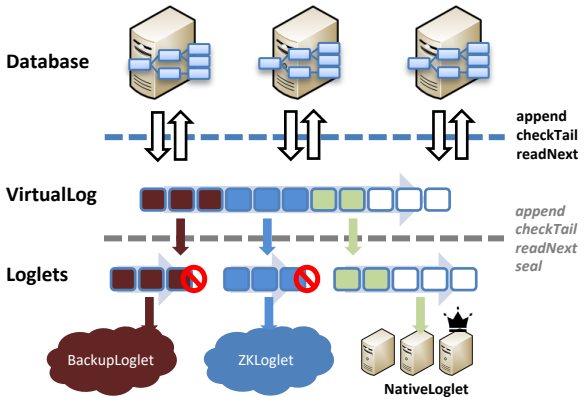


Figure 1: *Virtual consensus: servers replicate state via a VirtualLog, which is mapped to underlying Loglets.*

chains multiple shared log instances (called *Loglets*) into a single shared log. Different Loglets in a VirtualLog can be instances of the same ordering protocol with different parameters, leadership, or membership (e.g., different instances of MultiPaxos [45]); they can be entirely distinct log implementations (e.g., Raft [39], LogDevice [3], Scalog [16], or CORFU [7]); or they can be simple log shims over external storage systems (e.g., ZooKeeper [20] or HDFS [43]). Virtualization enables *heterogeneous reconfiguration*: a single VirtualLog can span different types of Loglets and dynamically switch between them.

We implemented virtual consensus in Delos, a database that stores control plane state for Facebook. Delos has been in production for over 18 months and currently processes over 1.8 billion transactions per day across all our deployments. One of its use cases is Twine's Resource Broker [44], which stores metadata for the fleet of servers in Facebook; each Delos deployment runs on 5 to 9 machines and manages server reservations for a fraction of the fleet. Internally, Delos is a shared log database [7, 8, 33]; it replicates state across servers by appending and playing back commands on the VirtualLog. Delos supports multiple application-facing APIs; we have a Table API in production, while a second ZooKeeper API is under development.

Virtual consensus in Delos *simplified the deployment and operation* of consensus implementations. Virtualization slashed time to deployment since we had the ability to deploy the system rapidly with an initial Loglet implementation, and later upgrade it without downtime. We reached production within eight months with a simple Loglet implemented as a shim over ZooKeeper (ZKLoglet); later, we obtained a 10X improvement in end-to-end latency in production by migrating online to a new, custom-built Loglet implementation (NativeLoglet). We also enabled seemingly infinite capacity for the VirtualLog by migrating older segments to a Loglet layered on cold storage (BackupLoglet); in turn, this allowed Delos to provide operators with a point-in-time restore capability. Loglets can be converged (i.e., collocated on the same machines as the database) or disaggregated (i.e., running on an entirely different set of machines), allowing operators to switch the Delos deployment mode on the fly to obtain different performance and fault-tolerance properties.

Virtual consensus also *simplifies the development* of new consensus implementations. Virtualization splits the complex functionality of consensus into separate layers: a control plane (the VirtualLog) that provides a generic reconfiguration capability; and a data plane (the Loglet) that provides critical-path ordering. While the VirtualLog's reconfiguration mechanism can be used solely for migrating between entirely different Loglet implementations, it can also switch between different instances of the same Loglet protocol with changes to leadership, roles, parameters, and membership. As a result, the Loglet itself can be a statically configured protocol, without any internal support for reconfiguration. In fact, the Loglet does not even have to implement fault-tolerant consensus (i.e., be highly available for appends via leader election), as long as it provides a fault-tolerant `seal` command, which is theoretically weaker and practically simpler to implement. When a Loglet fails for appends, the VirtualLog seals it and switches to a different Loglet, providing leader election and reconfiguration as a separate, reusable layer that can work with any underlying Loglet.

Accordingly, new Loglets are simple to design and implement since they are not required to implement fault-tolerant consensus or reconfiguration. To demonstrate this point, we describe the Delos NativeLoglet, which uses a primary-driven protocol that is unavailable for appends if the primary fails, but can support seals as long as a quorum is alive. New Loglets are also easy to construct via RAID-like composition; for example, we describe StripedLoglet, a thin shim layer that stitches together multiple Loglets to enable behavior equivalent to rotating sequencers [35] and sharded acceptors [7, 16].

Virtual consensus has some limitations. The reusability of VirtualLog-driven reconfiguration comes with a latency hit for certain types of reconfigurations such as planned leader changes. Loglets can optimize for specific cases by relying on their own in-band reconfiguration instead of the VirtualLog. A second limitation relates to generality: since we virtualize a specific API for consensus that captures a total order of commands, we do not currently support protocols that construct partial orders based on operation commutativity [26, 36]. In future work, we plan to extend virtual consensus to partially ordered shared logs [33].

We are the first to propose a virtualized shared log composed from heterogeneous log implementations. Prior work composes a logical shared log directly from storage servers [7, 9, 16, 47]; or virtualizes in the opposite direction, multiplexing homogenous streams over a single shared log [8, 48]. Delos is the first replicated database that can switch its consensus implementation on the fly to different

protocols, deployment modes, or codebases. While the theory of consensus has always allowed learners and acceptors to be disaggregated, Delos is also the first production system that can switch between converged and disaggregated acceptors.

In this paper, we make the following contributions:

- We propose virtualizing consensus via the novel VirtualLog and Loglet abstractions; and describe Delos, a storage system that implements these abstractions.
- Using production data, we show Delos upgrading to NativeLoglet without downtime for a 10X latency improvement.
- Using experiments, we show that Delos can: A) switch to a disaggregated Loglet for a 10X improvement in throughput under a 15ms p99 SLA; B) double its failure threshold without lowering throughput via a StripedLoglet that rotates sequencers. Further, we show that StripedLoglet can support over a million 1KB appends/s on a log-only workload by sharding acceptors.

2 The Path to Virtual Consensus

Virtualization for faster deployment: In 2017, Facebook needed a table store for its core control plane services with strong guarantees on durability, consistency, and availability. Two practical imperatives drove the design and development of this system: fast deployment (it had to reach production within 6-9 months) and incremental evolution (it had to support better performance over time).

At the time, Facebook already operated four different storage systems: a ZooKeeper service; a shared log service based on LogDevice [3]; a key-value service called ZippyDB [5]; and a replicated MySQL service [13]. None of these systems fit the exact use case, either due to a mismatch in API (e.g., ZooKeeper does not provide a table API) or fault-tolerance guarantees (e.g., the MySQL service provided inadequate availability).

Further, these systems could not be easily modified to provide the required API or guarantees. Each of them was a monolith: the database API could not be easily changed, nor could the underlying consensus protocol be re-used. In some systems, no abstraction boundary existed between the database and the consensus protocol. In other systems, an abstraction boundary did exist in the form of a shared log API, allowing the underlying consensus protocol to be reused; however, no support existed to migrate from one implementation of the abstraction to another.

Building yet another monolithic system from scratch – including a new consensus implementation – was not feasible since we had to hit deployment within 6-9 months. Layering the system over an existing shared log such as LogDevice would allow us to reach production quickly, but also tie us for perpetuity to the fault-tolerance and performance properties of that consensus implementation.

Our solution was to virtualize consensus. In the remainder of this paper, we describe how virtual consensus allowed us to reach production quickly with an existing implementation of consensus, and then migrate without downtime to new implementations.

Virtualization for faster development: Beyond fast initial deployment and online migration, virtualization also enabled faster development of new consensus implementations. On its own, the shared log abstraction simplifies consensus-based systems, separating applications from the logic of consensus via a data-centric API. Virtualizing the shared log further splits the consensus protocol into two layers: a control plane, which includes the logic for reconfiguration, and a data plane, which orders commands on the critical path.

In practice, such separation allowed us to incrementally improve the system by re-implementing just the data plane of the consensus protocol via new Loglets, while reusing the VirtualLog control plane for features such as leader election and membership changes. In the process, we completely changed the operational characteristics and performance of the system, as we describe later.

Importantly, such a separation also enables diversity in the data plane. The last few years have seen a number of consensus protocols with novel ordering mechanisms [3, 7, 14–16, 22, 24, 31, 35, 37, 39, 42], providing vastly different trade-offs between performance and fault-tolerance. By making it easier to implement such protocols and deploy them within running systems, virtualization lowers the barrier to innovation.

3 Abstractions for Virtual Consensus

In this paper, we propose virtualizing consensus by virtualizing the shared log abstraction. We have three design goals for virtualization. First, virtualization should be *transparent to applications*, which should be unmodified and oblivious to the virtualized nature of the log. Second, virtualization should allow underlying logs to be *simple and diverse*, lowering the barrier to new log implementations. Third, virtualization should allow for *migration* between implementations without downtime. We obtain these properties via two core abstractions.

In virtual consensus, the application operates above a VirtualLog, which stitches together multiple independent Loglets. The VirtualLog and Loglets expose a conventional shared log API (see Figure 2). Applications can `append` an entry, receiving back a log position; call `checkTail` to obtain the first unwritten position; call `readNext` to read the first entry in the passed-in range; and call `prefixTrim` to indicate that a prefix of the log can be trimmed. Virtualization requires two additions to this basic API: a `seal` command, which ensures that any new appends fail; and an augmented `checkTail` response

```
class ILoglet {
        logpos_t append(Entry payload);
        pair<logpos_t,bool> checkTail();
        Entry readNext(logpos_t min, logpos_t
            max);
        logpos_t prefixTrim(logpos_t trimpos);
        void seal();
}
class IVirtualLog : public ILoglet {
        bool reconfigExtend(LogCfg newcfg);
        bool reconfigTruncate();
        bool reconfigModify(LogCfg newcfg);
}
```

Figure 2: *Loglet and VirtualLog APIs.*

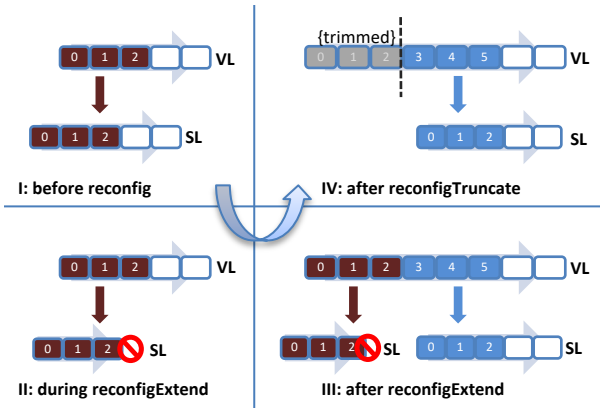


Figure 3: *The VirtualLog reconfigures from a single Loglet (I) by first sealing the Loglet (II) and installing a new Loglet (III). Later, the old Loglet is removed (IV).*

that indicates via a boolean whether the log is sealed. In addition, the VirtualLog also implements extra reconfiguration APIs to add and remove Loglets.

The VirtualLog is the only required source of fault-tolerant consensus in the system, providing a catch-all mechanism for any type of reconfiguration. The Loglet does not have to support fault-tolerant consensus, instead acting as a pluggable data plane for failure-free ordering. Existing systems typically struggle to implement monolithic consensus protocols that are simple, fast, and fault-tolerant. In virtual consensus, we divide and conquer: consensus in the VirtualLog is simple and fault-tolerant (but not necessarily fast, since it is invoked only on reconfigurations), while consensus in the Loglet is simple and fast (but not necessarily fault-tolerant). We now describe these abstractions and their interaction in detail.

3.1 The VirtualLog abstraction

The VirtualLog implements a logical shared log by chaining a collection of underlying Loglets. In this section, we use the term 'client' to refer to an application process that accesses the VirtualLog. Clients accessing the VirtualLog see a shared, append-only virtual address space that is strongly consistent (i.e., linearizable [18]), failure-atomic, and highly available. Internally, this address space is mapped to the individual address spaces of different Loglets in a chain. Operations to the VirtualLog are translated to operations on underlying Loglets based on this chain-structured mapping.

The simplest possible VirtualLog is a trivial singleton chain: $[0,\infty)$ of the VirtualLog is mapped to $[0,\infty)$ of a single Loglet. In this case, commands to the VirtualLog are passed through unmodified to the underlying Loglet. A more typical chain consists of multiple Loglets, mapping different segments of the virtual address space to each log: for example, $[0,100)$ is mapped to $[0,100)$ of Loglet A; $[100,150)$ is mapped to $[0,50)$ of Loglet B; $[150,\infty)$ to $[0,\infty)$ of C. We use the following notation for such a chain: $[0 \xrightarrow{A} 100 \xrightarrow{B} 150 \xrightarrow{C} \infty]$.

Any `append` and `checkTail` commands on a VirtualLog are directed to the last Loglet in the chain, while `readNext` commands on a range are routed to the Loglet storing that range. In the process, log positions are translated from the virtual address space to each Loglet's individual address space (for example, in the chain described above, if an `append` that is routed to Loglet C returns position 10, we map it to position 160 on the VirtualLog). Log positions can be contiguous (i.e., every position has an entry) or sparse (i.e., positions can be left unoccupied), depending on the underlying Loglet. Importantly, only the last log in the chain is appendable (we call this the active segment); the other logs are sealed and return errors on appends (we call these sealed segments).

The VirtualLog can be reconfigured to a new chain via its API (see Figure 2). The `reconfigExtend` call changes the active segment of the VirtualLog so that new appends are directed to an entirely different Loglet. Reconfigurations can also modify sealed segments via the `reconfigModify` call (e.g., to replace failed servers within a sealed Loglet). The `reconfigTruncate` call is used to remove the first sealed segment from the chain (e.g., when the VirtualLog's address space is trimmed).

3.2 VirtualLog design

The VirtualLog is composed of two distinct components: a client-side layer exposing the shared log API, which routes operations to underlying Loglets based on the chain-structured mapping; and a logically centralized metadata component (MetaStore) that durably stores the chain. Each client maintains a local (potentially stale) cached copy of the chain.

The MetaStore component has a simple API: it is a single versioned register supporting a conditional write. Reading the MetaStore returns a value with an attached version. Writing

to it requires supplying a new value and an expected existing version.

The primary technical challenge for the VirtualLog is providing clients with a shared, strongly consistent, and highly available virtual address space. In steady-state, when the chain remains unchanged, this task is trivial: the client-side layer can use its locally cached copy of the chain to route operations. However, the chain can be changed via the VirtualLog reconfiguration APIs shown in Figure 2.

Any client can initiate a reconfiguration, or complete a reconfiguration started by some other client. Reconfiguration involves three steps: *sealing* the old chain, *installing* the new chain on the MetaStore, and *fetching* the new chain from the MetaStore. The following reconfiguration protocol is expressed entirely in the form of data-centric operations against the Loglet and MetaStore; it assumes nothing about the internal implementation of either component.

Step 1: Sealing the current chain by sealing its last Loglet: The first step of a reconfiguration involves sealing the current chain (C_i) to stop new appends from succeeding within it. To seal the current chain, the reconfiguring client simply calls `seal` on the active segment, since this is the only segment that receives appends, and all other segments are already sealed. Seals on a Loglet are idempotent; accordingly, multiple clients can concurrently seal the current chain. Once the current chain is sealed by the reconfiguring client, any subsequent `append` on the current chain by a client returns an error. After sealing the active segment, the client calls `checkTail` to retrieve its tail; this determines the start of the new active segment.

Step 2: Installing the new chain in the MetaStore: Once the old chain C_i is sealed, the reconfiguring client writes a new chain C_{i+1} to the MetaStore. The MetaStore is simply a versioned register supporting a conditional write; accordingly, it only accepts the new chain C_{i+1} if the existing chain is C_i. In effect, multiple reconfiguring clients – after running step 1 idempotently – can race to install the new chain in the MetaStore, with at most one guaranteed to win. The chain is stored as a list of segments with start/stop positions, each with an opaque Loglet-specific configuration.

Step 3: Fetching the new chain from the MetaStore: In the final step, the reconfiguring client fetches the latest chain from the MetaStore. In the common case, this step can be omitted if the reconfiguring client succeeded in installing its candidate chain in Step 2. Alternatively, if the write in Step 2 failed, some other client may have won the race and installed a different chain, which we have to fetch.

Concurrency: After a client seals a chain in Step 1, it is possible that other clients continue operating within it. An `append` to the VirtualLog using the sealed chain will be routed to its last Loglet, which is now sealed in the new chain. As a result, a client issuing appends in the old chain will obtain an error code indicating that the Loglet is sealed; it will then fetch the latest chain from the MetaStore and retry.

A `checkTail` to the VirtualLog using the sealed chain also gets routed to the last Loglet in the chain. In response, the Loglet returns not just its tail position, but also a bit indicating whether or not it is sealed. If the Loglet has been sealed, then the client knows that it is operating on a stale chain, which means in turn that the computed tail is likely to not be the true tail of the VirtualLog. In this case, it fetches the latest chain from the MetaStore and retries.

Failure Atomicity: When a client encounters a sealed chain, it is possible that it does not find a newer chain in the MetaStore. This can happen if the reconfiguring client (which sealed the chain) either failed or got delayed after the seal step but before installing the new chain. In this case, after a time-out period, the client 'rolls forward' the reconfiguration by installing its own new chain. Note that the client completing the reconfiguration does not know the original intention of the failed client (e.g., if it was reconfiguring to a different Loglet type); hence, it creates a default new chain by cloning the configuration of the previous active segment.

Reconfiguration Policy: The protocol above provides a generic mechanism for reconfiguration. However, it has to be invoked based on some policy. There are three primary drivers of reconfiguration. First, planned reconfigurations (e.g., upgrading to a faster Loglet) are driven via a command line tool by operators. Second, the VirtualLog calls `reconfigTruncate` on itself when it trims the entirety of its first sealed Loglet while servicing a `prefixTrim`. For example, if the application calls `prefixTrim(100)` on chain $[0 \xrightarrow{A} 100 \xrightarrow{B} \infty]$; the VirtualLog trims all of A and then reconfigures to chain $[100 \xrightarrow{B} \infty]$. Third, individual Loglets that do not implement their own leader election or reconfiguration are responsible for detecting failures and requesting a `reconfigExtend` on the VirtualLog, as we describe later.

A subtle point is that an old chain only has to be sealed if it conflicts with a newer chain: i.e., the new chain remaps some unwritten virtual address to a different Loglet. Reconfigurations for sealed segments (e.g., to rebuild failed servers in a sealed Loglet, or to copy and remap a sealed segment to a different Loglet) do not change the locations of unwritten virtual addresses. As a result, they do not necessarily require the old chain to be sealed first before the new chain is installed; different clients can concurrently operate in the old and new chains. Similarly, truncation (i.e., removing the first segment) does not require the old chain to be sealed; if a client with the old chain tries to access an address on a truncated Loglet, it may succeed or receive an error saying the entry has been deleted. In practice, this means rebuild and GC activity on sealed segments does not interfere with appends at the tail of the VirtualLog.

3.3 The VirtualLog MetaStore

As described above, the VirtualLog stores a mapping – its chain of constituent Loglets – in a MetaStore: a single ver-

sioned register supporting a conditional write predicated on the current version. We now discuss the requirements and design space for this component.

The VirtualLog MetaStore is a necessary and sufficient source of fault-tolerant consensus in our architecture; as we describe later, Loglets are not required to implement consensus. The MetaStore has to be highly available for writes; accordingly, it requires a fault-tolerant consensus protocol like Paxos. Since the VirtualLog (and its MetaStore) is a separate, reusable layer, we need to implement this fault-tolerant consensus protocol only once. Further, the MetaStore is not required to be particularly fast, since it is accessed by the VirtualLog only during reconfigurations.

Why does the VirtualLog require its own MetaStore? Existing reconfigurable systems often store similar information (e.g., the set of servers in the next configuration) inline with the same total order as other commands (within the last configuration [34] or a combination of the old and new configurations [39]). In this case, the steps of sealing the old chain and writing the membership of the new chain can be done in a single combined operation. In the VirtualLog, this would be equivalent to storing the identity of the next Loglet within the current active Loglet while sealing it. However, such a design requires the Loglet itself to be highly available for writes (i.e., implement fault-tolerant consensus), since reconfiguring to a new Loglet would require a new entry to be written to the current Loglet. With a separate MetaStore, we eliminate the requirement of fault-tolerant consensus for each Loglet. Since one of our design goals is to make Loglets simple and diverse, we choose to use a separate MetaStore.

Using a separate MetaStore means the common-case latency of a reconfiguration consists of a `seal`, a `checkTail`, and a write to the MetaStore. In our current setting (control plane applications running within a single data center), reconfiguration latencies of 10s of ms are tenable. If reconfiguration is driven by failure, the latency of failure detection is typically multiple seconds in any case, to avoid false positives [30]. In the future, when we run across regions, it may be important to optimize for planned reconfiguration (e.g., replacing servers); since the Loglet is still available in this case, we can potentially reconfigure by storing inline commands within the Loglet itself, borrowing existing techniques such as α-windows [25, 34].

3.4 The Loglet abstraction

The Loglet is the data plane abstraction in virtual consensus: a shared log designed to operate as a segment of the VirtualLog. The requirements for a Loglet are minimal: it provides totally ordered, durable storage via the shared log API. Significantly, the Loglet can operate within a static configuration; it does not have to provide support for role or membership changes. It does not have to support leader election, either; i.e., it is not required to provide high availability for `append` calls. Instead, the Loglet provides a highly available `seal` command that prevents new appends from being successfully acknowledged. The VirtualLog uses such a sealing capability to support highly available `append` calls via reconfiguration, as described earlier in this section.

A `seal` bit does not require fault-tolerant consensus. Compared to similar data types that are equivalent to consensus, it differs from a write-once register [42], since only one 'value' can be proposed (i.e., the bit can be set); and a sticky bit [40], since it has only two states (undefined and set) rather than three. It can be implemented via a fault-tolerant atomic register that stores arbitrary values [6, 11], which in turn is weaker than consensus and not subject to the FLP impossibility result [17]. As we describe later, a `seal` is also much simpler to implement than a highly available `append`.

In addition to supporting `seal`, the Loglet provides an augmented `checkTail` to return its sealed status along with the current tail (i.e., `checkTail` returns a $(tailpos, sealbit)$ tuple rather than a single tail position). To lower the burden of implementing this extra call on each Loglet, it is designed to have weak semantics: the tail position and seal status do not need to be checked atomically. Instead, the `checkTail` call is equivalent to a conventional checkTail and a checkSeal executed in parallel, combined in a single call for efficiency.

In similar vein, a successful `seal` call ensures that any new `append` call will not be successfully acknowledged to the appending client; however, these failed appends can become durable and be reflected by future `checkTail` calls on the Loglet due to specific failure scenarios. As a result, calling `checkTail` on a sealed log can return increasing values for the tail position even after the log is successfully sealed. These 'zombie' appends on a sealed log do not appear on the VirtualLog's address space, which maps to fixed-size segments of the Loglet's address space (i.e., if the VirtualLog chain is $[0 \xrightarrow{A} 100 \xrightarrow{B} \infty]$, appends on log A can become durable beyond 100 without any impact on the VirtualLog). These semantics are sufficient to implement a linearizable VirtualLog: all we need is that any `append` on a sealed log throws an exception, and that any `checkTail` returns the seal status correctly.

The Loglet API provides a common denominator interface for different log implementations. Such implementations may provide availability for appends via internal leader election protocols; they may even support their own reconfiguration protocols for adding and removing storage servers. In such cases, the VirtualLog can be used to reconfigure across different Loglet types, while each Loglet can perform its own leader election and membership changes. To draw an analogy with storage stacks, Loglets can be functionally simple (e.g., like hard disks) or rich (e.g., like SSDs).

The log positions returned by a Loglet can be contiguous or sparse, depending on its internal implementation. Loglets that implement their own leader election or reconfiguration protocols typically expose sparse address spaces, since the log

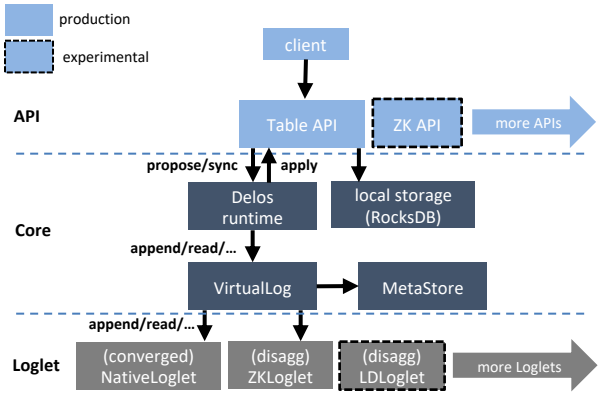


Figure 4: *Delos design: multiple APIs can run over a Core platform, which in turn can switch dynamically between multiple Loglet implementations.*

position often embeds some notion of a membership epoch.

In the case where the Loglet operates within a static configuration and relies on the VirtualLog for any form of reconfiguration, it reacts to failures by invoking `reconfigExtend` on the VirtualLog, which seals it and switches to a different Loglet. In this case, each Loglet is responsible for its own failure detection, and for supplying a new Loglet configuration minus the failed servers. In other words, while the VirtualLog provides the mechanism for reconfiguration, Loglets are partially responsible for the policy of reconfiguration.

4 Delos: Design and Implementation

Delos is a replicated storage system for control plane services. The design of Delos is driven by a number of requirements unique to control plane services: low dependencies on other services; strong guarantees for durability, availability, and consistency; and rich and flexible storage APIs. Delos is a fully replicated ACID database providing strict serializability. It does not implement transparent sharding or multi-shard transactions, but can be layered under other systems that provide these capabilities. Delos occupies a similar role in the Facebook stack to Google's Chubby [12], or the open-source etcd [2] and ZooKeeper [20].

The Delos database is similar to Tango [8]. Each Delos server maintains a local copy of state in RocksDB and keeps this state synchronized via state machine replication (SMR) over the VirtualLog. When a server receives a read-write transaction, it serializes and appends it to the VirtualLog without executing it. The server then synchronizes with the VirtualLog; when it encounters a transaction in the log (whether appended by itself or other servers), it executes the operation within a single thread as a failure-atomic transaction on its local RocksDB. The transaction returns when the appending server encounters it in the VirtualLog and applies it to the local RocksDB store. To perform a read-only transaction, the server first checks the current tail of the VirtualLog (obtaining a linearization position); it then synchronizes its local state with the VirtualLog until that position. The read-only transaction is then executed on the local RocksDB snapshot. For efficiency, Delos borrows a technique from Tango, queuing multiple read-only transactions behind a single outstanding synchronization with the VirtualLog.

As Figure 4 shows, the logic described above is separated into three layers on each Delos server: an API-specific wrapper at the top; a common Core consisting of a runtime that exposes an SMR interface, which in turn interacts with the VirtualLog; and individual Loglets under the VirtualLog. This layered design provides extensibility in two dimensions. First, Delos can support multiple Loglets under the VirtualLog, which is the focus of this paper. Second, Delos can support multiple application-facing APIs on a single platform. Each API wrapper is a thin layer of code that interacts with the Delos runtime and provides serialization logic against RocksDB. We support a Table API in production, with support for transactions, range queries, and secondary indices; we are currently deploying a second API identical to ZooKeeper. The ability to support multiple APIs on a common base is not novel: most state machine replication libraries treat the application as a black box. However, Delos provides a larger subset of application-agnostic functionality within the common Core, including local durable storage, backups, and state transfer when new servers join.

4.1 The Delos VirtualLog

In Delos, the VirtualLog is implemented via a combination of a client-side library and a MetaStore implementation. The library code implements the protocol described in Section 3.2, interacting with underlying Loglet implementations and the MetaStore. Initially, Delos went to production with the MetaStore residing on an external ZooKeeper service as a single key/value pair. Later, to remove this external dependency, we implemented an embedded MetaStore that runs on the same set of Delos servers as the database layer and VirtualLog library code.

To implement this embedded MetaStore, we used Lamport's construction of a replicated state machine from the original Paxos paper [25], which uses a sequence of independent Paxos instances. Each such instance is a simple, unoptimized implementation of canonical single-slot Paxos, incurring two round-trips to a quorum for both writes and reads. As in Lamport's description, each Paxos instance stores the membership of the next instance. We further simplify the protocol by disallowing pipelined writes at each proposer, and removing liveness optimizations such as leader election across multiple slots.

Such a protocol has inadequate latency and throughput to be used in the critical path of ordering commands, which is why

Loglet	Consensus	Deployment	Prod	Use
ZK	Yes	Disagg	Yes	Bootstrap
Native	No	Conv/Disagg	Yes	Primary
Backup	Yes	Disagg	Yes	Backup
LogDevice	Yes	Disagg	No	Perf
Striped	No	Conv/Disagg	No	Perf

Figure 5: *Different Loglet implementations in Delos.*

so many (complex) variants of Multi-Paxos exist. However, it is more than sufficient for a control plane protocol that is invoked only for reconfigurations.

4.2 The Delos Loglet(s)

Loglets can be converged, running on the Delos database servers; or disaggregated, as shims on the Delos servers accessing an external service. Each deployment model has its benefits: a converged log allows Delos to operate without a critical path service dependency, and without incurring the extra latency of accessing an external service. Disaggregation enables higher throughput by placing the log on a storage tier that can be independently scaled and I/O-isolated from the database servers. Delos currently supports three disaggregated Loglets (see Figure 5): ZKLoglet stores log entries on a ZooKeeper namespace; LogDeviceLoglet is a pass-through wrapper for a LogDevice service; BackupLoglet layers over an HDFS-like filesystem service used for cold storage. All three backing systems – ZooKeeper, LogDevice, and the HDFS-like filesystem – internally implement fault-tolerant consensus, including leader election and reconfiguration; Delos uses the VirtualLog solely to switch to/from them.

4.2.1 Loglets sans consensus: NativeLoglet

We argued earlier that Loglets can be simple since they do not require fault-tolerant consensus (i.e., highly available appends) or any form of reconfiguration. We now describe the NativeLoglet, which illustrates this point. A NativeLoglet can be either converged or disaggregated; we describe its converged form, which is how we use it in production.

Each Delos server – in addition to running the materialization logic and the VirtualLog code – runs a NativeLoglet client and a NativeLoglet server (or LogServer). One of the Delos servers also runs a sequencer component. The NativeLoglet is available for `seal` and `checkTail` as long as a majority of LogServers are alive; and for `append` if the sequencer is also alive. Each LogServer stores a local on-disk log, along with a seal bit; once the seal bit is set, the LogServer rejects new appends to its local log. The local log stores commands in a strictly ascending order that can have gaps (i.e., it may not store every command).

We first define some terms before describing the protocol. A command is *locally committed* on a particular LogServer after it has been written and synced to its local log. The *local tail* for a particular LogServer is the first unwritten position in its local log. A command is *globally committed* once it is locally committed on a majority of LogServers and all previous commands have been globally committed. The *global tail* of the NativeLoglet is the first globally uncommitted log position. The NativeLoglet does not have gaps; i.e., every position from 0 up to the global tail stores a globally committed command. Each component (i.e., LogServers, clients, and the sequencer) maintains a *knownTail* variable: the global tail it currently knows about, which can trail the actual global tail. Components piggyback *knownTail* on the messages they exchange, updating their local value if they see a higher one.

append: To append commands to the NativeLoglet, Delos servers send requests to the sequencer. The sequencer assigns a position to each command and forwards the request to all LogServers. It considers the append globally committed (and acknowledges to the client) once it receives successful responses from a majority of unsealed LogServers. If a majority of LogServers report that they have been sealed, an error is returned indicating that the NativeLoglet is sealed. In all other cases where a majority of LogServers respond negatively or fail to respond before a timeout, the sequencer retries the append. Retries are idempotent (i.e., the same command is written to the same position), and the sequencer continues to retry until the append succeeds or the NativeLoglet is sealed.

Each LogServer locally commits a particular log position n only after the previous position $n-1$ has either (1) been locally committed on the same server, or (2) has been globally committed on a majority of servers (i.e., $knownTail > n-1$). The sequencer maintains an outgoing queue of appends for each LogServer, and omits sending a command to a particular LogServer if it knows the command has already been globally committed. As a result, slow LogServers do not block appends from completing on other LogServers and a trailing LogServer can catch up more easily because it is allowed to omit storing commands that are already globally committed.

seal: Any client can seal the NativeLoglet by contacting each LogServer to set its seal bit. If the seal completes successfully – i.e., a majority of LogServers respond – future appends are guaranteed to return an error code indicating the NativeLoglet is sealed. Note that a successful seal operation can leave different LogServers with different local tails.

checkTail: This call returns both the current global tail of the NativeLoglet, as well as its current seal status. Any client can issue a `checkTail` by sending a command to all the LogServers and waiting for a majority to respond. Once a majority responds, the `checkTail` call follows a simple 5-state state machine, as described in Figure 6. For ease of exposition, we assume that no more than a majority responds; if this is not true, the protocol below can work trivially by discarding the extra responses, though in practice we use the additional infor-

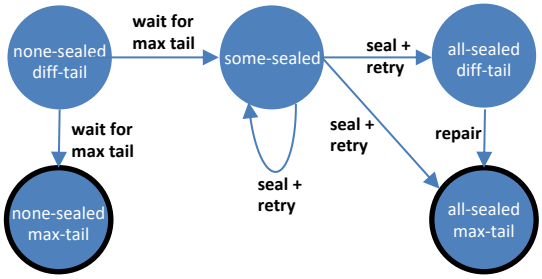


Figure 6: *NativeLoglet* `checkTail` *state machine.*

mation for better performance. Three outcomes are possible for the returned seal bits:

1. all-sealed: In the case where all responding LogServers are sealed and they all return the same local tail X, the return value is (X, true). However, it is possible that the LogServers can have different local tails (e.g., if a seal arrives while an append is ongoing). In this case, the client 'repairs' the responding LogServers to the maximum returned position X_{max}, copying over entries from the LogServers that have them (and bypassing the seal bit). It then returns X_{max} to the application. Note that repair is safe: the single sequencer ensures that there can never be different entries for the same position on different LogServers. This repair activity can result in the 'zombie' appends described in Section 3.4, where appends on a sealed Loglet are not acknowledged but can complete later due to repairs.

2. some-sealed: In the case where the responding LogServers have a mix of sealed and unsealed status bits, the client issues a `seal` first and then reissues the `checkTail`. In the absence of an adversarial failure pattern (e.g., where the seal continually lands on a different majority), the subsequent `checkTail` should return the *all-sealed* case above where all responding LogServers are sealed.

3. none-sealed: In the case where none of the responding LogServers are sealed, the client picks the maximum position X_{max} and then waits for its own *knownTail* to reach this position. While waiting, if the client discovers that some LogServer is sealed, the `checkTail` is in the *some-sealed* state described above, and proceeds accordingly. If the sequencer fails, the client's *knownTail* may never reach X_{max}; in this case, the Loglet will eventually be sealed, putting the client in the *some-sealed* or *all-sealed* state.

The latency of the `checkTail` in the *none-sealed* case depends on how quickly the client's *knownTail* is updated, along with its knowledge of the seal status of a majority of LogServers. Clients quickly and efficiently discover this information via an extra API exposed by each LogServer, which allows them to ask for notification when the local tail reaches a particular position or the LogServer is sealed.

readNext: Loglet semantics dictate that `readNext` behavior is defined only for log positions before the return value of a previous `checkTail` call from the same client. As a result, a `readNext` call translates to a read on a particular log position that is already known to exist. This simplifies the `readNext` implementation: the client first checks the locally collocated LogServer to find the entry. If it can't find the entry locally, it issues a read to some other LogServer. Note that a quorum is not required for reads, since we already know that the entry has been committed; we merely have to locate a copy.

To reiterate, the NativeLoglet does not implement fault-tolerant consensus: it becomes unavailable for appends if the sequencer fails. As a result, the `append` path has no complex leader election logic. Instead, the NativeLoglet implements a highly available `seal`, which is a trivial operation that sets a bit on a quorum of servers. The `checkTail` call follows a compact state machine for determining the seal status and global tail of the NativeLoglet. Practically, we found this protocol much easier to implement than fault-tolerant consensus: it took just under 4 months to implement and deploy a production-quality NativeLoglet.

When the sequencer or one of the LogServers fails, the NativeLoglet is responsible for detecting this failure and invoking reconfiguration on the VirtualLog (which in turn seals it and switches to a new NativeLoglet). In our implementation, we use a combination of in-band detection (e.g., the sequencer detecting that it has rebooted, or that other servers are persistently down) and out-of-band signals (via a gossip-based failure detector, as well as information from the container manager) to trigger reconfiguration. In other words, the VirtualLog provides the mechanism of reconfiguration / leader election, while the NativeLoglet handles the policy by selecting the LogServers and sequencer of the new NativeLoglet instance.

4.2.2 Loglets via composition: StripedLoglet

The StripedLoglet stripes a logical address space across multiple underlying Loglets. The mapping between address spaces is a simple and deterministic striping: logical position L_0 maps to position A_0 on stripe A; L_1 maps to position B_0; L_2 to C_0; L_3 to A_1; and so on (see Figure 7).

Incoming `append` calls to the StripedLoglet are routed to individual stripe Loglets in round-robin order. This routing is done independently at each StripedLoglet client (i.e., the Delos database servers). When the append on an individual Loglet returns with a stripe-specific position (e.g., A_1), we map it back to a logical position (e.g., L_3). However, we do not acknowledge the append to the StripedLoglet client immediately; instead, we wait until all prior logical positions have been filled, across all stripes. This ensures linearizability for the StripedLoglet: if an append starts after another append completes, it obtains a greater logical position. For example, in Figure 7, an `append` is routed to stripe B at position B_2 or L_7; but it is not acknowledged or reflected by `checkTail` until L_6 appears on stripe A at position A_2.

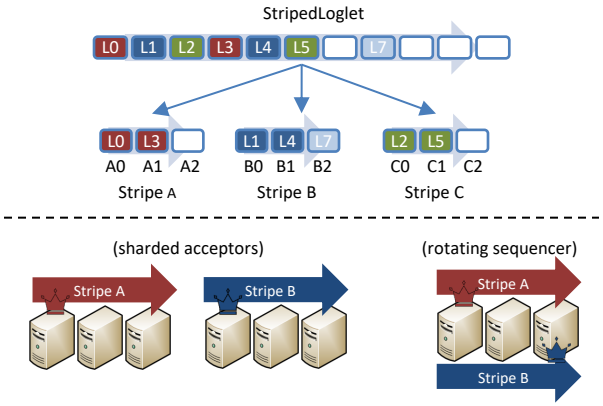


Figure 7: *The StripedLoglet stripes over underlying Loglets. Loglets can be disjoint (sharded acceptors) or have overlapping membership but different sequencers (rotating sequencer).*

A `checkTail` call fans out to all stripes and returns the first unfilled logical position, while `readNext` calls are directed to the corresponding stripe. For this protocol to work, the StripedLoglet requires the stripe Loglets to have contiguous log positions. While the NativeLoglet provides this property, LogDeviceLoglet does not: LogDevice embeds an epoch number (generated by its own internal reconfiguration mechanism) within the log position, so that positions are typically contiguous but can skip forward if an internal reconfiguration takes place.

We found composition to be a simple and effective way to create protocols with new properties. The StripedLoglet is a shim layer with only around 300 lines of code and consists entirely of invocations on the underlying Loglets; yet it provides a versatile building block for scaling throughput. We experimented with two uses of it (shown in Figure 7):

Rotating sequencer: A StripedLoglet can be composed from NativeLoglets with identical LogServers but different sequencers. For instance, a 9-node NativeLoglet will bottleneck on the single sequencer, which has to transmit each entry 8 times. Instead, a StripedLoglet can be layered over two NativeLoglet stripes, each of which uses the same LogServers but a different sequencer.

Sharded acceptors: A StripedLoglet can be layered over multiple disaggregated Loglets, achieving a linear scaling of throughput similar to CORFU [7] or Scalog [16], albeit via a design that doesn't require a centralized sequencer or separate ordering layer. StripedLoglet also differs by relying on virtualization: it implements a Loglet API over other Loglets. As a result, StripedLoglet can scale any existing Loglet while inheriting its fault-tolerance properties (i.e., the StripedLoglet fails if any of its stripes fail).

Note that the StripedLoglet code is identical for both these use cases: what changes is the composition of the individual Loglets. These Loglets can have different memberships (and even entirely different Loglet implementations) in the sharded acceptor case; or identical membership (and the same Loglet implementation) but different sequencers in the rotating sequencer case.

From the viewpoint of the VirtualLog, the StripedLoglet is like any other Loglet; it has to be sealed as a whole (i.e., every stripe has to be sealed) even if only one of its stripes needs to be changed via the VirtualLog reconfiguration mechanism. In the future, we plan to explore schemes for selectively sealing and replacing a single stripe.

5 Evaluation

We use two hardware setups for evaluating Delos. The first is the *production* hardware that most of our instances run on, which consists of 5 servers with shared boot SSDs. Since Delos has to run in a variety of data centers, we cannot assume specific or dedicated storage hardware. The second is *benchmark* hardware with dedicated NVMe SSDs. In both setups, we run within Twine [44] containers, and have production-grade debug logging and monitoring enabled.

We show numbers for two workloads. The first is real production traffic. For a representative deployment, the workload consists of 425 queries/sec and 150 puts/sec on the Delos Table API. Write size has a median of 500 bytes and a max of 150KB. Each deployment stores between 1GB and 10GB. In production, Delos takes local snapshots every 10 minutes and ships them to a backup service every 20 minutes.

The second workload is a synthetic one consisting of single-key puts and gets. The value consists of a single 1KB blob. The keys are chosen from an address space of 10M keys; we select keys randomly with a uniform distribution and generate random values, since this provides a lower bound for performance by reducing caching and compression opportunities, respectively. We pre-write the database before each run with 10GB; this matches our production data sizes.

Delos runs with two Loglets in production: ZKLoglet and NativeLoglet. In our experiments, we additionally use LogDeviceLoglet (or LDLoglet) and StripedLoglet. The external ZooKeeper service used by ZKLoglet lives on a set of 5 servers similar to our production hardware, running on shared boot SSDs and collocated with other jobs. LDLoglet uses a LogDevice service deployed on a set of 5 servers similar to our benchmark hardware, with dedicated NVMe SSDs.

In all the graphs, we report 99th percentile latency over 1-minute windows. We assume a p99 SLA of 15ms, which matches our production requirements.

5.1 The Benefit of Virtualization

Virtual consensus allowed Delos to upgrade its consensus protocol in production without downtime. In Figure 8, we show the actual switch-over happening from ZKLoglet to

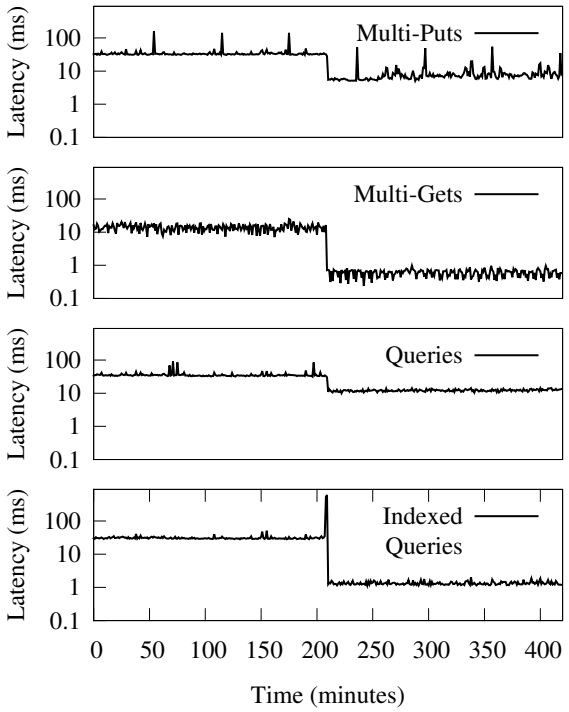


Figure 8: *Production upgrade from ZKLoglet to NativeLoglet (log-scale y-axis).*

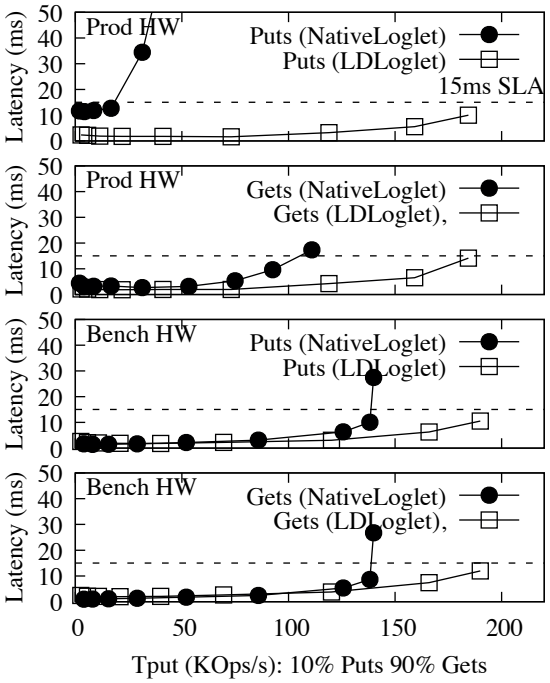


Figure 9: *Delos can scale throughput between 10X (top) and 33% (bottom) for different HW types with a disaggregated LDLoglet instead of a converged NativeLoglet.*

a converged NativeLoglet for the first time on a Delos production instance on April 2nd 2019. Switching to a different log implementation provides substantially lower latency for a production workload. The graph shows p99 latencies for four categories of Table operations: we see 10X improvements for multi-gets and indexed queries, and a 5X improvement for multi-puts. Additionally, the switch-over happens without downtime: p99 latency spikes for indexed queries during reconfiguration, but otherwise service availability is not disrupted. The latency improvement is largely due to the unoptimized nature of our ZKLoglet implementation, which simply writes a new sequential ZooKeeper key on each append.

Interestingly, the graph shows two reconfigurations: the first is a `reconfigExtend` that seals the ZKLoglet and switches the active segment to a NativeLoglet, causing the visible shift in performance; the second, which happens a few minutes later, is a `reconfigTruncate` that removes the old ZKLoglet segment from the VirtualLog, but does not require a seal (as described in Section 3.2) and hence does not cause any disruption. The hourly spikes in multi-puts are due to periodic large writes from the application.

Delos can scale throughput by running over a disaggregated Loglet. In Figure 9, we plot throughput on the x-axis and p99 latency on the y-axis, for the synthetic workload with 90% gets and 10% puts. We compare the converged NativeLoglet vs. the disaggregated LDLoglet. In the top two graphs, the Delos database runs on production HW with shared SSDs; latency with NativeLoglet starts rising at 15K ops/s for puts due to contention between the Loglet and the database. With a disaggregated LDLoglet running on 5 other machines, throughput scales 10X higher at 150K ops/s without breaching 15ms p99 latency for either puts or gets. This 10X improvement is partly due to more HW (twice the machines); better HW for the log (LDLoglet runs over dedicated NVMe SSDs); and less I/O contention (the database and log are on different machines).

In the bottom two graphs, Delos runs on benchmark HW with dedicated SSDs; the performance hit for the converged NativeLoglet is less stark due to more IOPS to share between the log and database, with latency rising for both puts and gets at around 139K ops/s. The disaggregated LDLoglet provides 33% higher throughput at 190k ops/s. For both types of HW, disaggregation allows the shared log to utilize a separate, dedicated set of resources. We get similar results running against a disaggregated NativeLoglet instead of LDLoglet, but wanted to highlight Delos' ability to run over different consensus implementations.

Delos can switch dynamically between converged and disaggregated modes without downtime. Figure 10 (Left) demonstrates the ability of Delos to change dynamically between converged and disaggregated modes, and the utility of doing so in order to handle workload shifts. In this experiment, we run the synthetic workload on the high-contention production HW. We want to maintain a 15ms p99 latency SLA.

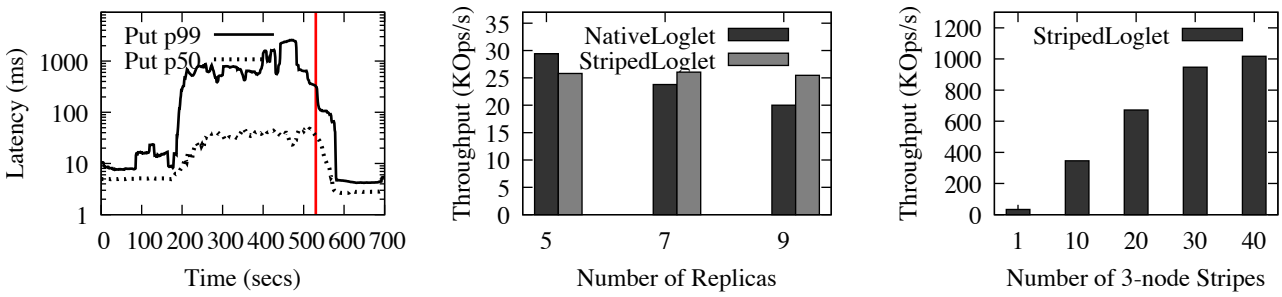


Figure 10: ***Left:*** *Delos can dynamically switch from converged (NativeLoglet) to disaggregated (LDLoglet) logging to handle workload shifts (log-scale y-axis).* ***Middle:*** *StripedLoglet (rotating sequencer) alleviates the sequencer bottleneck as Delos scales.* ***Right:*** *StripedLoglet (sharded acceptors) scales to 1M+ appends/s for a log-only workload.*

For the first 180 secs, we run a constant, low workload of 100 puts/sec; after that, we increase the workload to 2500 puts/sec. Delos initially runs over the NativeLoglet, which meets the 15ms SLA for the low workload. But when the workload switches, Delos+NativeLoglet is no longer able to keep up due to I/O contention for the SSDs, with p99 latency degrading to over a second. At around 530 secs, we reconfigure to use LDLoglet; this reduces I/O pressure on the local SSDs, allowing p99 latencies to drop back to under 15ms (after a 60-second lag due to the 1-minute sliding window).

If a disaggregated log provides better throughput and lower latency, why not always use one? First, disaggregation is inefficient from a resource standpoint for low workloads, using 10 machines compared to 5 with a converged log. Second, converged Delos does not depend on any external service in the critical path, which is important for some control plane applications.

New protocols with useful properties can be implemented via Loglet composition. In the NativeLoglet, all appends are routed via the sequencer node. For a 100% 1KB put workload on a 5-node cluster, Delos is bottlenecked by the IOPS of the NativeLoglet LogServers. However, when we run Delos over 9 LogServers for higher fault-tolerance, the bottleneck shifts to the NativeLoglet sequencer, which now has to send out each entry 8 times. If we instead use a StripedLoglet over 2 NativeLoglets (each with the same set of LogServers but different sequencers), we rotate the sequencing load across two servers. As Figure 10 (Middle) shows, Delos+StripedLoglet runs 25% faster than Delos+NativeLoglet with 9 nodes on the benchmark HW.

We also ran log-only experiments with StripedLoglet in Figure 10 (Right). We created a StripedLoglet over multiple 3-node NativeLoglets, and appended 1KB payloads from 20 VirtualLog clients. We see linear scaling of throughput as we go from 1 stripe (3 LogServers) to 30 stripes (i.e., 90 LogServers); beyond that, our clients became the bottleneck. The LogServers run on shared NVMe SSDs, which provide 30K IOPS with a p99 of 75ms; we report the maximum throughput for each configuration with a p99 latency of under 75ms. We obtained similar results with 4KB payloads (750K appends/s with 30 shards); this is the highest reported single-log throughput in a non-emulated experiment, exceeding CORFU (570K/s) and Scalog (255K/s). Delos cannot leverage such a high-throughput log, since it bottlenecks on log playback; we plan to explore selective playback [8], as well as compare against Scalog's higher emulated numbers.

5.2 The Cost of Virtualization

Virtualization is inexpensive in terms of critical path latency. In most cases, the VirtualLog acts as a simple pass-through layer. Figure 11 (Left) shows the p99 latency of VirtualLog and NativeLoglet operations; this data is measured over a one-hour time period on a production cluster running over the NativeLoglet. For `append` and `checkTail`, virtualization adds 100-150 μseconds to p99 latency; this is largely due to the overhead of an asynchronous Future-based API. In contrast, `readNext` is a synchronous pass-through call and adds only a few μseconds.

Reconfigurations occur within 10s of ms. In Figure 11 (Middle), we show a histogram of all reconfigurations in a 1-month period on our production clusters. Since `reconfigTruncate` does not call `seal`, it has lower latency than `reconfigExtend`. For our applications, reconfiguration latencies of 10s of ms are tenable. The vast majority of these reconfigurations are triggered by 1) continuous deployment of software upgrades; and 2) machine preemptions for hardware maintenance, kernel upgrades, etc. Actual failures constitute a small percentage of the reconfigurations. In practice, clusters see a few reconfigurations per day; for example, one of our production clusters was reconfigured 98 times in the 1-month period.

Virtualization does not affect peak throughput. We performed an apples-to-apples comparison of Delos to ZooKeeper on our benchmark HW. We translate puts into SetData commands and gets into GetData commands on the

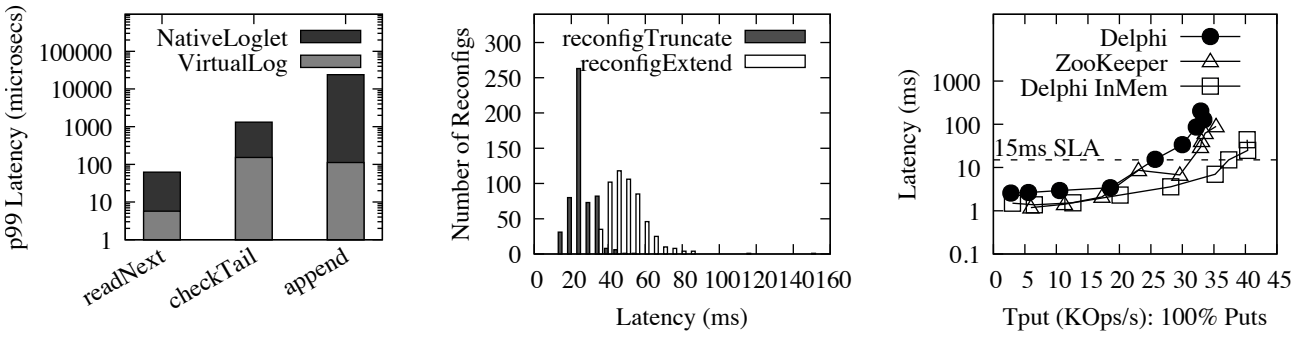


Figure 11: ***Left:*** *virtualization overhead is low in production: 6 µs for readNext and 100 to 150 µs for append / checkTail p99.* ***Middle:*** *reconfigurations take tens of ms in production.* ***Right:*** *Delos+NativeLoglet matches ZooKeeper performance.*

ZooKeeper API. Since ZooKeeper does not support more than a few GB of data, we ran with a 1GB database. Figure 11 (Right) shows that ZooKeeper can provide over 30K puts/sec before p99 latency degrades beyond 15ms. In contrast, Delos+NativeLoglet manages around 26K puts/sec. The primary reason for the performance difference is that ZooKeeper stores its materialized state in memory while Delos uses RocksDB. We also ran a version of Delos where the materialized state lives in memory; this prototype hit 40K puts/sec. While storing state in RocksDB causes a performance hit at small sizes, it enables us to scale; the Delos+NativeLoglet curve for a 100 GB database (not shown in the graph) is nearly identical to the 1GB case. These results show that Delos performance is comparable to unvirtualized systems.

6 Discussion

Virtual consensus provided a number of ancillary benefits for the operation of Delos.

Fate-sharing... but only when my fate is good: In production, Delos typically runs as a converged database with no external dependencies, where all logic and state (including the database and the log) resides on a set of 5 machines. However, the database / learner layer is simultaneously more fragile than the log / acceptor layer (since it is updated frequently to add features) and requires lower fault-tolerance (only one learner needs to survive, compared to a quorum of acceptors). If two converged replicas crash out of five, another failure can cause unavailability and data loss for the log. In this situation (which was not uncommon), we found it valuable to reconfigure the system to a disaggregated log, temporarily decoupling the fate of the database and the log. Once the database was restored to five replicas, we reconfigured back. This style of temporary decoupling also proved valuable when we discovered latent bugs in the NativeLoglet; we reconfigured to ZKLoglet, rolled out hotfixes, and then reconfigured back. Currently, switching between converged and disaggregated logs is a manual operation driven by operators; in the future, we may explore automated switching.

Consensus is forever... until it's not: Deletion of arbitrary entries is typically quite difficult in conventional consensus protocols. However, with virtual consensus, we can delete an entry simply by changing the metadata of the VirtualLog. Similarly, altering written entries is possible via remapping. We found this kind of surgical editing capability useful when faced with site-wide outages: on one occasion, a "poison" entry caused hangs on all learners processing the log.

ZooKeeper over Delos... over ZooKeeper: Virtualization often enables interesting and practically useful layerings; for example, it is routine in storage stacks to run a filesystem over a virtual block device that in turn is stored on a different instance of the same filesystem. Virtual consensus brings similar flexibility to replicated databases: in our current stack, we have the ability to run our experimental ZooKeeper API over the VirtualLog, which in turn can run over the ZooKeeper-based ZKLoglet.

7 Related Work

Virtual consensus builds upon a large body of work in fault-tolerant distributed systems. Most approaches to reconfigurable replication (including Viewstamped Replication [32, 38], ZAB [22], Raft [39], Vertical Paxos [27], and SMART [34]) use protocols for leader election and reconfiguration that are tightly intertwined with the ordering protocol. Virtual Synchrony [10, 46] is an exception: it uses a unified view change protocol for leader election and reconfiguration that sits above the ordering mechanism used within each view. This unified approach is also found in more recent systems such as Derecho [21] and CORFU [7]. Reconfiguration has been explored as a layer above a generic state machine by Stoppable Paxos [28, 29]; unlike Loglets, the underlying state machine has to implement fault-tolerant consensus.

Virtual consensus borrows many ideas from this literature, combines them, and applies them to a production system: for example, unified leader election and reconfiguration (Vir-

tual Synchrony); a segmented total order with potentially disjoint configurations (SMART); an external auxiliary (Vertical Paxos); and a generic ordering abstraction (Stoppable Paxos). However, virtual consensus also differs from all this prior work in several important aspects. First, we target and demonstrate diversity in the ordering layer (e.g., deploying new layers, switching to disaggregated mode, etc.). Second, the ordering layer is only required to provide a highly available seal, which is weaker than fault-tolerant consensus and easier to implement. Finally, virtual consensus applies and extends these ideas to shared logs, which pose unique opportunities (e.g., data-centric APIs that hide complexity) and challenges (e.g., application-driven trims and explicit reads).

To assess the generality of the Loglet abstraction, we did an informal survey of recent replication protocols. The majority of these protocols either directly expose a log API [3,7,16,37,42] or can be wrapped as a Loglet [14,15,22,24,31,35,39]. Virtualization gives us the ability to easily experiment with these protocols under Delos and deploy them to production. Other work – such as protocols that exploit speculation [41] or commutativity [4,26,36] – does not currently fit under the Loglet API.

Virtual consensus is based on the shared log approach for building replicated systems; we leverage the existence of the shared log API as a boundary between the database and consensus mechanism. Shared logs were first introduced by CORFU [7] and Hyder [9] as an API for consensus. Subsequently, CorfuDB [1] was the first production database to be deployed over a shared log API. Along similar lines, systems such as Kafka [23] and LogDevice [3] have become popular in industry, exposing large numbers of individual, independently ordered logs. In contrast, research has largely focused on scaling a single log, either via faster ordering protocols [16] or different forms of selective playback [8,48]. Rather than build a faster shared log or a more scalable database above it, virtual consensus seeks to make such systems simpler to build and deploy as they become commonplace in industry.

8 Conclusion

Virtual consensus enables faster deployment and development of replicated systems. Reconfiguration and leader election is encapsulated in a control plane (the VirtualLog) that can be reused across any data plane (Loglets). Delos is the first system that supports heterogeneous reconfiguration, allowing changes to not just the leader or the set of servers in the system, but also the protocol, codebase, and deployment model of the consensus subsystem. As a result, new systems can be developed and deployed rapidly (e.g., Delos hit production within 8 months by leveraging an external service for its Loglet); and upgraded without downtime to provide significantly different performance and fault-tolerance properties (e.g., we hot-swapped Loglets in production for a 10X latency reduction).

Acknowledgments

We would like to thank our shepherd, Jay Lorch, and the anonymous OSDI reviewers. Many people contributed to the Delos project, including Adrian Hamza, Mark Celani, Andy Newell, Artemiy Kolesnikov, Ali Zaveri, Ben Reed, Denis Samoylov, Grace Ko, Ivailo Nedelchev, Mingzhe Hao, Maxim Khutornenko, Peter Schuller, Suyog Mapara, Rajeev Nagar, Russ Arun, Soner Terek, Terence Feng, and Vidhyashankar Venkataraman. Marcos Aguilera, Jose Faleiro, Dahlia Malkhi, and Vijayan Prabhakaran provided valuable feedback on early iterations of this work.

References

[1] CorfuDB. https://github.com/corfudb.

[2] etcd. https://etcd.io/.

[3] LogDevice. https://logdevice.io/.

[4] Ailijiang, A., Charapko, A., Demirbas, M., and Kosar, T. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems 31*, 1 (2019), 211–223.

[5] Annamalai, M., Ravichandran, K., Srinivas, H., Zinkovsky, I., Pan, L., Savor, T., Nagle, D., and Stumm, M. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *Proceedings of USENIX OSDI 2018*.

[6] Attiya, H., Bar-Noy, A., and Dolev, D. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM) 42*, 1 (1995), 124–142.

[7] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., and Davis, J. D. CORFU: A Shared Log Design for Flash Clusters. In *USENIX NSDI 2012*.

[8] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of ACM SOSP 2013*.

[9] Bernstein, P. A., Das, S., Ding, B., and Pilman, M. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *Proceedings of ACM SIGMOD 2015*.

[10] Birman, K. P., and Joseph, T. A. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems (TOCS) 5*, 1 (1987), 47–76.

[11] BURKE, M., CHENG, A., AND LLOYD, W. Gryff: Unifying Consensus and Shared Registers. In *Proceedings of USENIX NSDI 2020*.

[12] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of USENIX OSDI 2006*.

[13] CAO, Z., DONG, S., VEMURI, S., AND DU, D. H. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of USENIX FAST 2020*.

[14] CHARAPKO, A., AILIJIANG, A., AND DEMIRBAS, M. PigPaxos: Devouring the communication bottlenecks in distributed consensus. *arXiv preprint arXiv:2003.07760* (2020).

[15] DANG, H. T., CANINI, M., PEDONE, F., AND SOULÉ, R. Paxos Made Switch-y. *ACM SIGCOMM Computer Communication Review 46*, 2 (2016), 18–24.

[16] DING, C., CHU, D., ZHAO, E., LI, X., ALVISI, L., AND VAN RENESSE, R. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *Proceedings of USENIX NSDI 2020*.

[17] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[18] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[19] HOWARD, H., MALKHI, D., AND SPIEGELMAN, A. Flexible Paxos: Quorum intersection revisited. In *Proceedings of OPODIS 2016*.

[20] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of USENIX ATC 2010*.

[21] JHA, S., BEHRENS, J., GKOUNTOUVAS, T., MILANO, M., SONG, W., TREMEL, E., RENESSE, R. V., ZINK, S., AND BIRMAN, K. P. Derecho: Fast State Machine Replication for Cloud Services. *ACM Transactions on Computer Systems (TOCS) 36*, 2 (2019), 1–49.

[22] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of IEEE DSN 2011*.

[23] KLEPPMANN, M., AND KREPS, J. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Engineering Bulletin, 38 (4)* (2015).

[24] KOGIAS, M., AND BUGNION, E. HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services. In *Proceedings of ACM EuroSys 2020*.

[25] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998), 133–169.

[26] LAMPORT, L. Generalized Consensus and Paxos. *Microsoft Research Technical Report MSR-TR-2005-33* (2005).

[27] LAMPORT, L., MALKHI, D., AND ZHOU, L. Vertical Paxos and Primary-Backup Replication. In *Proceedings of ACM PODC 2009*.

[28] LAMPORT, L., MALKHI, D., AND ZHOU, L. Stoppable Paxos. *Microsoft Research Technical Report (unpublished)* (2008).

[29] LAMPORT, L., MALKHI, D., AND ZHOU, L. Reconfiguring a State Machine. *SIGACT News 41*, 1 (2010), 63–73.

[30] LENERS, J. B., WU, H., HUNG, W.-L., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the FALCON spy network. In *Proceedings of ACM SOSP 2011*.

[31] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. Just say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of USENIX OSDI 2016*.

[32] LISKOV, B., AND COWLING, J. Viewstamped Replication Revisited. In *MIT Technical Report* (2012).

[33] LOCKERMAN, J., FALEIRO, J. M., KIM, J., SANKARAN, S., ABADI, D. J., ASPNES, J., SEN, S., AND BALAKRISHNAN, M. The FuzzyLog: a Partially Ordered Shared Log. In *Proceedings of USENIX OSDI 2018*.

[34] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART Way to Migrate Replicated Stateful Services. In *Proceedings of ACM EuroSys 2006*.

[35] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of USENIX OSDI 2008*.

[36] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of ACM SOSP 2013*.

[37] NAWAB, F., ARORA, V., AGRAWAL, D., AND EL ABBADI, A. Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments. In *Proceedings of EDBT 2015*.

[38] OKI, B. M., AND LISKOV, B. H. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of ACM PODC 1988*.

[39] ONGARO, D., AND OUSTERHOUT, J. K. In Search of an Understandable Consensus Algorithm. In *Proceedings of USENIX ATC 2014*.

[40] PLOTKIN, S. A. Sticky Bits and Universality of Consensus. In *Proceedings of ACM PODC 1989*.

[41] PORTS, D. R., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of USENIX NSDI 2015*.

[42] SHIN, J.-Y., KIM, J., HONORÉ, W., VANZETTO, H., RADHAKRISHNAN, S., BALAKRISHNAN, M., AND SHAO, Z. WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems. In *Proceedings of ACM SoCC 2019*.

[43] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proceedings of IEEE MSST 2010*.

[44] TANG, C., YU, K., VEERARAGHAVAN, K., KALDOR, J., MICHELSON, S., KOOBURAT, T., ANBUDURAI, A., CLARK, M., GOGIA, K., CHENG, L., CHRISTENSEN, B., GARTRELL, A., KHUTORNENKO, M., KULKARNI, S., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., VENKATESAN, V., AND ZHANG, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of USENIX OSDI 2020*.

[45] VAN RENESSE, R., AND ALTINBUKEN, D. Paxos Made Moderately Complex. *ACM Computing Surveys (CSUR) 47*, 3 (2015), 1–36.

[46] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A Flexible Group Communication System. *Communications of the ACM 39*, 4 (1996), 76–83.

[47] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *ACM SIGMOD 2017*.

[48] WEI, M., TAI, A., ROSSBACH, C. J., ABRAHAM, I., MUNSHED, M., DHAWAN, M., STABILE, J., WIEDER, U., FRITCHIE, S., SWANSON, S., ET AL. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *USENIX NSDI 2017*.



Byzantine Ordered Consensus without Byzantine Oligarchy

Yunhao Zhang,† Srinath Setty,⋆ Qi Chen,⋆ Lidong Zhou,⋆ and Lorenzo Alvisi†
†*Cornell University* ⋆*Microsoft Research*

Abstract

The specific order of commands agreed upon when running state machine replication (SMR) is immaterial to fault-tolerance: all that is required is for all correct deterministic replicas to follow it. In the permissioned blockchains that rely on Byzantine fault tolerant (BFT) SMR, however, nodes have a stake in the specific sequence that ledger records, as well as in preventing other parties from manipulating the sequencing to their advantage. The traditional specification of SMR correctness, however, has no language to express these concerns. This paper introduces *Byzantine ordered consensus*, a new primitive that augments the correctness specification of BFT SMR to include specific guarantees on the total orders it produces; and a new architecture for BFT SMR that, by factoring out ordering from consensus, can enforce these guarantees and prevent Byzantine nodes from controlling ordering decisions (a Byzantine oligarchy). These contributions are instantiated in Pompē,[1] a BFT SMR protocol that is guaranteed to order commands in a way that respects a natural extension of linearizability.

1 Introduction

This paper aims to add a new dimension to state machine replication (SMR) [62], a fundamental building block in fault-tolerant distributed computing, by introducing a way to express, reason about, and enforce specific properties about how the SMR protocol *orders* the commands it receives.

SMR coordinates a set of replicas of a deterministic service so that, collectively, they implement the abstraction of a single, correct server. In particular, the protocol sequences client-issued requests to produce a total order, which correct replicas then follow when processing the requests. As long as the system includes sufficiently many correct replicas, voting on replica outputs guarantees that clients of the replicated service can recognize and accept only output values that would have been generated by a correct server.

SMR totally orders client requests by running an instance of consensus for each position in the request sequence. The only requirement on the sequence agreed upon is that it eventually contains all requests from correct clients. Indeed, if all SMR is used for is fault-tolerance, no further legislation is necessary: the specific sequence of states that correct replicas traverse is immaterial.

[1]The urban ritual of the pompē (πομπή, or *procession*) was central to civic and religious life in the Byzantine empire.

Not so, however, when SMR is used (typically, in a Byzantine fault tolerant (BFT) configuration) across multiple administrative domains to support a blockchain. Consider, for instance, permissioned blockchains like Libra [3], CCF [60], or HyperLedger Fabric [6]: the specific order of transactions held by their ledger can have significant financial implications [24, 47]. The nodes running these protocols are not just interested in converging on an agreed-upon ledger: they have a real stake in the specific sequence that ledger records, as well as in preventing other parties from manipulating the sequencing to their advantage.

The traditional specification of correctness for (BFT) SMR, however, has no language for addressing such concerns; because it attaches no significance to the sequence it produces, it is intrinsically incapable of characterizing what makes a total order "right" or "wrong".

Our aim in this paper is to introduce a framework for expressing and enforcing such distinctions. A key challenge is that the lack of expressiveness in the correctness specification of SMR has deep architectural roots. Specifically, in standard leader-based SMR [16, 43], the ordering of a command is hardcoded in the protocol that adds the command to the ledger: the leader runs concurrently a set of consensus instances, each dedicated to filling a specific ledger position with a command of its choosing.

Thus, we pursue a two-pronged approach: for expressiveness, we expand the correctness specification of the BFT SMR primitive; for enforcement, we articulate a new architecture for BFT replication that makes it possible to implement in practice our newly-introduced correctness requirements.

Our first contribution is to introduce *Byzantine ordered consensus*, a new primitive that augments the correctness specification of BFT SMR to include the enforcement of specific guarantees on the total orders it produces. The new specification allows the nodes that implement a replicated state machine to associate an *ordering indicator* to the commands they ultimately agree upon. Through these indicators, nodes can express how they would like commands to be ordered with respect to one another. The correctness conditions for Byzantine ordered consensus specify, given a set of input ⟨ordering indicator, command⟩ pairs, the set of allowable total orders for the commands.

To identify meaningful correctness conditions in the presence of Byzantine nodes, we draw inspiration from classic work in social choice theory [7, 9, 11] and explore the limits of what *can* be guaranteed in the presence of Byzantine

nodes. In particular, we ask: is it possible to prevent Byzantine nodes from dictating the ordering of commands? And, at the other end of the spectrum, is it possible to completely prevent Byzantine nodes from wielding influence on that order? We find that, while eliminating Byzantine influence is provably impossible, new mechanisms can prevent the establishment of a Byzantine oligarchy.

Simply expressing these correctness conditions, however, is not enough: we need the means to enforce them. Our second main contribution is to introduce a new general architecture for BFT protocols that *factors ordering out of consensus*: it cleanly separates the process of establishing the relative order of commands from the consensus step necessary to add those ordered commands to the ledger. This separation completely eliminates the leader's ability to control a command's position in the ledger; at the same time, it retains the simplicity and efficiency of having a leader in charge of the consensus step.

Finally, we design, implement, and evaluate Pompē, a BFT SMR protocol based on Byzantine ordered consensus that enforces *ordering linearizability*, a new correctness condition that prevents a Byzantine oligarchy and offers correct nodes a meaningful guarantee about the order ultimately recorded in the ledger. Informally, it ensures that if the *lowest* timestamp that any correct node assigns to command c_2 is larger than the *highest* timestamp that any correct node assigns to c_1, then c_1 will precede c_2 in the ledger.

We implement Pompē by extending prior state-of-the-art BFT implementations [1, 2]. Our experimental evaluation demonstrates that while Pompē incurs higher latencies than its baselines, Pompē can achieve higher throughput at competitive latencies by batching commands in both the ordering step and the consensus step. For example, with $n = 4$ nodes in a single datacenter, a version of Pompē that extends ordering linearizability to HotStuff [2, 67] achieves a throughput of approximately 360,000 commands/s at a latency of about 53 ms, which corresponds to 40% higher throughput and 6% higher latency than HotStuff. Additionally, since in Pompē nodes can order multiple commands in parallel, we find that, if the computing resources assigned to each node are scaled up proportionally with the number of nodes, Pompē can sustain its high throughput in settings with 100 nodes distributed over three geo-distributed datacenters.

2 Background and motivation

The increasing popularity of blockchains as a platform for cooperation and data sharing among mutually distrustful parties has brought about a renewed interest in Byzantine fault tolerance (BFT). In particular, permissioned blockchains, which promise a platform for executing commands without trusting a centralized authority, have adopted as their core the standard BFT SMR architecture [62]. Transitioning BFT to this new application domain has introduced some new challenges. One that has received much attention is scalability. Traditional BFT SMR protocols have typically targeted deployments involving a number of nodes in the single digits, while some permissioned blockchains envision running BFT at scales that are two orders of magnitude or larger. A new breed of BFT SMR protocols have raised to this challenge, finding clever ways to pipeline requests and streamline the communication required to achieve consensus [10, 31, 50, 54, 68].

In this paper we address a different challenge that emerges when applying BFT SMR in a blockchain context, one fundamental enough to bring into question whether this primitive is sufficiently expressive to serve as the basis for this new class of applications.

Consider the correctness specification of SMR: it requires all correct nodes replicating a service to traverse the same set of states and produce the same outputs. If replicas are deterministic, an expedient way to satisfy this requirement is to ensuring that all correct replicas process the same sequence of inputs: identical inputs translate into identical states and outputs. As long as these inputs are valid client commands, the correctness specification assigns no semantic meaning to the particular order in which they are executed by the replicas: that order is simply a syntactic mechanism used to achieve the desired safety property.

In blockchains, however, the specific order adopted by the underlying SMR protocol tends to have rich semantic implications, which often translate into substantially different financial rewards for the parties involved. Allowing some users to front-run their commands ahead of others clearly gives them an unfair advantage in applications such as auctions and exchanges [47, 57]; indeed, a recent paper [24] details how bots have reaped from unsuspecting parties profits in excess of $6M by replicating, within the Ethereum network, transaction manipulation strategies already notorious in traditional exchanges [47]. Yet, order manipulation (including censorship, selective inclusion, command reordering, and command injection) does not, per se, violate the specification of SMR, the technology at the core of projects like Libra [3].

Unfortunately, adding the "BFT" qualifier to SMR does not help address these concerns: all it does is to ensure that the standard SMR specification continues to hold even if some nodes are Byzantine. The crux, rather, is that the correctness expectations of blockchain users do not stop at requiring all ledgers to hold the same total order: *which order* matters.

A symptom of the discomfort caused by this semantic gap is the growing focus on curbing the discretion of the single node that, in Paxos-like BFT SMR, leads the consensus decisions: if Byzantine, this *leader* node can single-handedly control the ledger's order. Proposed solutions include rotating leaders [13, 21, 68]; holding leaders accountable for their actions [33, 35]; or developing outright "leaderless" protocols that give no node a special role in the execution of consensus [23, 44, 54]. These efforts are a step in the right direction, but they also, arguably, miss the point. While it is clear enough that leaving a single leader in full control of the ledger's order is undesirable, they fiddle with a low-level mechanism with-

out offering a way to express the correctness guarantees that such mechanisms, whatever they may be, should enforce. Recent work on order-fairness [38], concurrent with ours, takes a further step forward by adding to consistency and liveness the additional requirement of *transactional order-fairness*; however, it offers neither a general framework for synthesizing desirable ordering guarantees from the ordering preferences of individual nodes, nor tries to precisely characterize the degree to which Byzantine nodes can affect ordering.

This paper argues that the correct approach to bridge the current semantic gap is instead to start from first principles. Thus, we introduce a new primitive, *Byzantine ordered consensus*, that expands the correctness specifications of BFT SMR so it can express specific correctness guarantees about the total orders it produces. Inspired by classic work in social choice theory [7, 9, 11], Byzantine ordered consensus lets participating nodes not only propose commands, but also indicate how they prefer to see them ordered. Essentially, Byzantine ordered consensus makes it possible to specify which total orders a correct BFT SMR is allowed to produce, given the nodes' ordering preferences. For example, assuming that nodes use as their ordering preference the time they first see a command, we show that it is possible to require total orders that satisfy a natural generalization of linearizability: *ordering linearizability*, which ensures that, if the highest timestamp from all correct nodes for command c_1 is lower than the lowest timestamp from all correct nodes for c_2, c_1 is ordered before c_2.

The design space for ordering properties that we explore is delimited by two overarching concerns. On the one hand, we want to curb as much as possible the clout of Byzantine nodes; on the other hand, we want to ensure that the preferences of correct nodes will carry weight in the final ordering.

These goals can sometimes align; in particular, when it comes to preventing Byzantine nodes from solely controlling the ledger's final ordering. As we noted above, the standard approach to BFT SMR allows a Byzantine leader to alone dictate which command commits in which consensus instance, independent of what other nodes prefer. We aim for, and define, guarantees (such as ordering linearizability) that prevent such Byzantine dictatorships. Indeed, we show that it is possible to rule out a Byzantine oligarchy, in which Byzantine nodes are jointly able to determine the ordering decisions, regardless of the correct nodes' ordering preferences.

Sometimes, however, we find these goals fundamentally at odds with one another: in particular, we find that ensuring that each correct node has a saying in the final order makes it impossible, in general, to completely prevent Byzantine nodes from influencing the final order. This is the price, if you wish, of operating in a Byzantine democracy.

3 Byzantine ordered consensus

Byzantine ordered consensus generalizes BFT SMR to expose the ordering aspect explicitly, but preserves the same system model, which consists of a distributed system of n nodes (with at most f Byzantine faults) that act as clients as well as servers: they both propose commands and execute them. This model simplifies our presentation without any loss of generality; we discuss how it relates to different real-world deployment scenarios in Section 8.

Ordering indicators. As in standard BFT SMR, nodes in Byzantine ordered consensus propose commands as inputs and output a consistent totally-ordered ledger. Unlike standard BFT SMR, each node associates a proposed command c with an *ordering indicator* o, which is metadata indicating the node's ordering preference for c, so proposals are of the form $\langle o, c \rangle$. Let $\mathcal{O}$ denote the set of ordering indicators; we define an *order-before relation* $\prec_o$ on $\mathcal{O} \times \mathcal{O}$ as follows: For any pair of proposals $\langle o_1, c_1 \rangle$ and $\langle o_2, c_2 \rangle$, where $o_1, o_2 \in \mathcal{O}$, $o_1 \prec_o o_2$ indicates a preference to order c_1 before c_2.

Examples of ordering indicators include timestamps, sequence numbers, and dependency sets or graphs. For timestamps (or sequence numbers), the order-before relation $\prec_o$ can simply be the $<$ relation on timestamps (or sequence numbers), while for dependency sets/graphs, $\prec_o$ can be the subset/subgraph relation on dependency sets or graphs.

Profiles, executions, and traces. We refer to a set of $\langle o, c \rangle$ proposals as a *profile*. Let $\mathcal{P}^i$ and $\mathcal{P}^{\mathcal{C}}$ denote, respectively, the set of proposals from node i and the set of proposals from all correct nodes. Given a command c, we say that $c \in \mathcal{P}^{\mathcal{C}}$ if and only if there exists a correct node i and an ordering indicator o, such that $\langle o, c \rangle \in \mathcal{P}^i$.

In an *execution*, correct nodes follow their prescribed protocol and input their proposals from $\mathcal{P}^{\mathcal{C}}$, whereas Byzantine nodes and the network are under the control of an adversary. For a given profile, an execution can produce different *traces*; each trace captures a single deterministic run of the protocol, recording the behavior of all nodes (both correct and Byzantine) as well as of the adversarial network. Although all traces of an execution take as input the same $\mathcal{P}^{\mathcal{C}}$, the content of the ledger on which correct nodes agree may be different for different traces, because of the actions of Byzantine nodes or the behavior of the network. But what is the degree to which Byzantine nodes can exert their influence on a given protocol? And what is the price to curb it?

The politics of Byzantium. A minimum guarantee that any protocol should offer is to make it impossible for Byzantine nodes to dictate the ordering of the ledger's entries. It is out of concern for ensuring this guarantee that recent work in BFT SMR has focused on limiting the leader node's discretion in making ordering decision. The formal structure offered by Byzantine ordered consensus allows us to move past the inadequacies of the current mechanisms used to drive consensus and focus instead on a precise characterization of what any such mechanism should guarantee. In particular, we capture the intuition that Byzantine nodes can dictate the ordering decisions through the notion of Byzantine oligarchy.

Byzantine Oligarchy. A protocol execution is subject to a *Byzantine oligarchy* if and only if, for all profiles of correct nodes $\mathcal{P}^{\mathcal{C}}$, for all pairs of commands c_1 and c_2 in $\mathcal{P}^{\mathcal{C}}$, there exists a trace for $\mathcal{P}^{\mathcal{C}}$ that results in c_1 before c_2 in the ledgers of correct nodes and another trace for $\mathcal{P}^{\mathcal{C}}$ that results in c_2 before c_1 in the ledgers of correct nodes.

Intuitively, this definition conveys that, in a Byzantine oligarchy, the actions of Byzantine nodes can determine the ordering of any two commands c_1 and c_2, regardless of the ordering indicators from correct nodes.

Can we do more, and completely eliminate any influence of Byzantine nodes over the ledger's final ordering? The framework offered by Byzantine ordered consensus allows us to prove that this target can be achieved only at the price of denying *correct* nodes a voice in the ordering decision. The intuition is simple: since in general it is impossible to distinguish a priori between correct and Byzantine nodes, a policy that enfranchises the first group necessarily also gives some influence to the second.

To formalize this intuition, we introduce two new notions. First, we express what it means for a protocol to allow the ordering preferences of its nodes to influence the ledgers' final total order. Note that, if a node can influence the outcome, then there will be some circumstances in which the node's preferences will actually *determine* the outcome. The second notion we introduce characterizes the impact of according such influence to a Byzantine node.

Free Will. We say that a protocol respects the nodes' *free will* if and only if (i) for all profiles of correct nodes $\mathcal{P}^{\mathcal{C}}$, there exists a trace for $\mathcal{P}^{\mathcal{C}}$, such that all commands in $\mathcal{P}^{\mathcal{C}}$ appear in the ledgers of correct nodes in the trace and (ii) there exist two profiles $\mathcal{P}_A$ and $\mathcal{P}_B$, such that, for all commands c_1 and c_2 that appear in both profiles, there exists a trace for $\mathcal{P}_A$ that results in c_1 before c_2 in the ledgers of correct nodes and there exists a trace for $\mathcal{P}_B$ that results in c_2 before c_1 in the ledgers of correct nodes.

Free will rules out (i) arbitrarily denying proposed commands and (ii) trivial and predetermined ordering mechanisms (e.g., ordering commands by their hash values) .

Byzantine Democracy. We say that a protocol upholds *Byzantine democracy* if and only if there exists a profile of correct nodes $\mathcal{P}^{\mathcal{C}}$, such that for all pairs of commands c_1 and c_2 in $\mathcal{P}^{\mathcal{C}}$, there exists a trace for $\mathcal{P}^{\mathcal{C}}$ that results in c_1 before c_2 in the ledgers of correct nodes and another trace for $\mathcal{P}^{\mathcal{C}}$ that results in c_2 before c_1 in the ledgers of correct nodes.

Unlike a Byzantine oligarchy, a Byzantine democracy gives Byzantine nodes sway over the final ledger only for *some* profiles of correct nodes, rather than all of them.

We are now ready to formulate a theorem that places fundamental limits to the degree to which it is possible to curb the influence of Byzantine nodes.

Theorem 3.1. *Free will* $\implies$ *Byzantine democracy.*

Proof. Consider the following $n+1$ profiles, where $\mathcal{P}_{\#1} = \mathcal{P}_A$ and $\mathcal{P}_{\#n+1} = \mathcal{P}_B$ and every node proposes the same commands (though, possibly, with different ordering preferences) in $\mathcal{P}_A$ and $\mathcal{P}_B$.

$$\begin{aligned}
\mathcal{P}_A = \mathcal{P}_{\#1} &= \mathcal{P}_A^1 \cup \mathcal{P}_A^2 \cup \ldots \cup \mathcal{P}_A^{n-1} \cup \mathcal{P}_A^n \\
\mathcal{P}_{\#2} &= \mathcal{P}_B^1 \cup \mathcal{P}_A^2 \cup \ldots \cup \mathcal{P}_A^{n-1} \cup \mathcal{P}_A^n \\
\mathcal{P}_{\#3} &= \mathcal{P}_B^1 \cup \mathcal{P}_B^2 \cup \ldots \cup \mathcal{P}_A^{n-1} \cup \mathcal{P}_A^n \\
&\ldots \\
\mathcal{P}_{\#n} &= \mathcal{P}_B^1 \cup \mathcal{P}_B^2 \cup \ldots \cup \mathcal{P}_B^{n-1} \cup \mathcal{P}_A^n \\
\mathcal{P}_B = \mathcal{P}_{\#n+1} &= \mathcal{P}_B^1 \cup \mathcal{P}_B^2 \cup \ldots \cup \mathcal{P}_B^{n-1} \cup \mathcal{P}_B^n
\end{aligned}$$

In profile $\mathcal{P}_{\#i}$, the proposals of the first $i-1$ nodes are the same as in as in $\mathcal{P}_B$; those of the other $n-i+1$ nodes are the same as in $\mathcal{P}_A$. Because free will (condition (ii)) holds, there is a trace for $\mathcal{P}_{\#1}$ for which the ledgers of correct nodes order c_1 before c_2, and a trace for $\mathcal{P}_{\#n+1}$ where instead they appear in the opposite order. And, also because free will (condition (i)) holds, for each index k, there exists a trace for $\mathcal{P}_{\#k}$, such that c_1 and c_2 appear in the final ledgers. Then, there must exist some index i for which the relative order of c_1 and c_2 switches when going from $\mathcal{P}_{\#i}$ to $\mathcal{P}_{\#i+1}$. Consider the the smallest such i. $\mathcal{P}_{\#i}$ and $\mathcal{P}_{\#i+1}$ only differ in what node i proposes: in $\mathcal{P}_{\#i}$ node i's proposals come from $\mathcal{P}_A$; in $\mathcal{P}_{\#i+1}$, they come from $\mathcal{P}_B$. Hence, by choosing whether to $\mathcal{P}_A^i$ or $\mathcal{P}_B^i$, node i determines the relative order of c_1 and c_2.

If i is Byzantine, then Byzantine democracy holds for the following correct profile:

$$\mathcal{P}^{\mathcal{C}} = \mathcal{P}_B^1 \cup \ldots \cup \mathcal{P}_B^{i-1} \cup \mathcal{P}_A^{i+1} \ldots \cup \mathcal{P}_A^n \qquad \square$$

The definition of Byzantine democracy makes clear that there exist some profiles that allow Byzantine nodes to control ordering decisions. A natural question then is: can we design protocols that, by construction, enforce guarantees that specify profiles on which Byzantine nodes can have no influence? And what would such properties look like? We address the second question next, leaving the answer to the first to Section 4.

Ordering properties. Since under standard BFT assumptions (Section 4) correct nodes are more than two thirds of the total (a supermajority!), the profiles less likely to be influenced by Byzantine nodes are intuitively those in which the voting preferences of correct nodes are aligned. We examine two natural ordering properties that one might want to see holding in such profiles; other definitions are possible.

The first requires that, if the ordering indicators of correct nodes are unanimous on how to relatively order two commands, their preference should be reflected in the final ledger.

Ordering unanimity: For all profiles of correct nodes $\mathcal{P}^{\mathcal{C}}$, for all commands c_1 and c_2 that appear in $\mathcal{P}^{\mathcal{C}}$ and in the

ledgers of correct nodes, if, for every correct node i, $\langle o_1, c_1\rangle \in \mathcal{P}^i \wedge \langle o_2, c_2\rangle \in \mathcal{P}^i \Rightarrow o_1 \prec_o o_2$, and there exists at least one correct node j, such that $\langle o_1, c_1\rangle \in \mathcal{P}^j \wedge \langle o_2, c_2\rangle \in \mathcal{P}^j$ holds, then c_1 must precede c_2 in the ledgers of correct nodes.

The second ordering property is inspired by linearizability [36], which orders a command c_1 before a command c_2 if the first ends before the second starts.

Ordering linearizability: For all profiles of correct nodes $\mathcal{P}^{\mathcal{C}}$, for all commands c_1 and c_2 in $\mathcal{P}^{\mathcal{C}}$ and in the ledgers of correct nodes, let $O_1 = \{o_1 | \langle o_1, c_1\rangle \in \mathcal{P}^{\mathcal{C}}\}$ and $O_2 = \{o_2 | \langle o_2, c_2\rangle \in \mathcal{P}^{\mathcal{C}}\}$, if $o_1 \prec_o o_2$ holds for all $o_1 \in O_1$ and $o_2 \in O_2$, then c_1 must precede c_2 in the ledgers of correct nodes.

Informally, the "lowest" and "highest" ordering indicators in O_1 (or O_2) indicate when c_1 (or c_2) start and end in the collective perception of correct nodes. Hence, by analogy with linearizability, if all ordering indicators in O_1 are lower than those in O_2, then c_1 is to be ordered before c_2.

Unfortunately, even when correct nodes are unanimous, their wishes are not guaranteed to come true. The issue again arises from the tension between the desire of giving a voice to every correct node and the inability to distinguish a priori between correct and Byzantine nodes.

Theorem 3.2. *No protocol can uphold both free will and ordering unanimity.*

Proof (sketch). Consider the four-node profile ($f = 1$) in Figure 1. It is an example of what classic social choice theory calls a Condorcet cycle [11, 22]: for any two commands c_i and c_{i+1} (modulo 4) , three nodes prefer the first before the second; the fourth begs to differ.

$$\begin{aligned}
\mathcal{P}^1 &= \{\langle 1, c_1\rangle, \langle 2, c_2\rangle, \langle 3, c_3\rangle, \langle 4, c_4\rangle\} \\
\mathcal{P}^2 &= \{\langle 1, c_2\rangle, \langle 2, c_3\rangle, \langle 3, c_4\rangle, \langle 4, c_1\rangle\} \\
\mathcal{P}^3 &= \{\langle 1, c_3\rangle, \langle 2, c_4\rangle, \langle 3, c_1\rangle, \langle 4, c_2\rangle\} \\
\mathcal{P}^4 &= \{\langle 1, c_4\rangle, \langle 2, c_1\rangle, \langle 3, c_2\rangle, \langle 4, c_3\rangle\}
\end{aligned}$$

FIGURE 1—A Condorcet cycle

Since any single node may be Byzantine, the requirement of ordering unanimity applies to all ordering preferences endorsed by at least three nodes—but in this example they form a *cycle*, and thus cannot be all satisfied. □

Like ordering unanimity, ordering linearizability also promises to respect the collective preferences of correct nodes; fortunately, unlike the former property, it *is* achievable. What allows ordering linearizability to escape the Condorcet cycle trap is a simple insight: it expresses ordering preferences in terms of real-time *happened before*, a relation that is inherently acyclical. Indeed, as we show next, it is not only achievable, but can be efficiently implemented.

4 Pompē

Pompē is a new protocol explicitly designed for Byzantine ordered consensus that preserves the same interface as a standard BFT protocol: clients propose commands and correct nodes reach consensus on a sequence of committed commands. In addition to satisfying the standard safety and liveness properties of BFT SMR, Pompē introduces an ordering phase for Byzantine ordered consensus and prevents Byzantine oligarchies by enforcing ordering linearizability.

A new architecture. Pompē's two-phase architecture is designed to mirror the decoupling of ordering from consensus made possible by the ordered consensus primitive. First, an *ordering phase* decides the total ordering of commands, "locking" the relative position among the commands proposed in this phase in a way that Byzantine nodes cannot alter; then, a *consensus phase* allows all correct nodes to agree on a stable prefix of the final sequence, following the total ordering decisions in the ordering phase, and to record it in the ledger. We refer to commands in the ledgers of correct nodes as *stable commands*. Note that, since the total order of commands that have completed the ordering phase cannot be changed during the consensus phase, it is again safe to put a single leader node in charge of finalizing consensus. Thus, Pompē can retain the performance benefits of leader-based BFT SMR without fears of enabling a Byzantine oligarchy.

System model. As in prior works in the BFT SMR literature, we consider a distributed system with a set of $n = 3f + 1$ nodes, where up to f nodes can be *Byzantine* (i.e., deviate arbitrarily from their prescribed protocol) and the rest are *correct*. We assume the existence of standard cryptographic primitives (unforgeable digital signatures and collision-resistant hash functions) and that cryptographic hardness assumptions necessary to realize these primitives hold. Furthermore, we assume that each node holds a private key to digitally sign messages, and that each node knows the public keys of other nodes in the system. We consider an adversarial network that can drop, reorder, or delay messages. However, for liveness properties, we assume that the network satisfies a weak form of synchrony [16, 27, 28]. Finally, we assume that each node has access to a timer, which produces monotonically increasing timestamps each time it is queried.

4.1 Protocol description

We now describe how Pompē instantiates each of the phases in our new architecture. Throughout the protocol, we assume that correct recipients of messages that are not well-formed (e.g., because they carry an incorrect signature) will drop them: we omit these actions in the interest of brevity.

(1) Ordering phase. Pompē uses timestamps as ordering indicators. To "lock" a position for a command in a total order, Pompē proceeds in two steps.

In the first step, a node N_i with a command c collects signed timestamps on c from a quorum of $2f + 1$ nodes. The median timestamp in the set of $2f + 1$ signed timestamps is the *assigned timestamp* for c, and it determines the position of c in the total order. Because there are at most f Byzantine

nodes, by picking the median value, the assigned timestamp is both upper- and lower-bounded by timestamps from correct nodes. This is the key observation that allows the protocol to achieve ordering linearizability.

To lock this position in the total order for c, in the second step N_i broadcasts c along with its assigned timestamp and waits for it to be accepted by a quorum $2f+1$ nodes (we explain below what it means for a command to be accepted). If a command c is accepted by a quorum of $2f+1$ nodes, c is not only guaranteed to be included in the totally-ordered ledgers of correct nodes, but also that its position in the ledgers is determined by the assigned timestamp of c. We refer to such commands as *sequenced*.

Local state. Each node maintains the following local data structures: (1) localAcceptThresholdTS, an integer, initialized to 0, that tracks what N_j believes to be, currently, the latest possible timestamp of any stable command in the ledger; (2) localSequencedSet, a set, initially empty, that tracks all commands that the node has accepted; (3) highTS, an n-sized vector of integers where highTS$[i]$, initialized to 0, stores the highest timestamp received from node N_i; and (4) highTSMsgs, an n-sized vector of messages where highTSMsgs$[i]$, initialized to *null*, stores the message signed by node N_i that carried the value currently stored in highTS$[i]$.

To complete our discussion of each node's local state, we first need to introduce a simple protocol that nodes use to update their timers.

The protocol. Let $\mathcal{T}$ be the $(f+1)^{th}$ highest timestamp in highTS. Because at most f nodes are Byzantine, $\mathcal{T}$ is upper-bounded by a timestamp from a correct node. Let each node reset its timer to $\mathcal{T}$ whenever $\mathcal{T}$ is higher than the current value of the local timer. Periodically, each node broadcasts its current value of $\mathcal{T}$ in a Sync message to indicate that all correct nodes can now set their timer to be $\mathcal{T}$ or higher. To prove to its recipients that the $\mathcal{T}$ value is valid, the Sync message also includes the sender's highTSMsgs vector. □

We are now ready to define two additional data structures: (4) globalSyncTS stores the highest $\mathcal{T}$ received so far in a Sync message; and (5) localSyncTS stores instead the node's local timestamp at the time it received that Sync message.

Actions. Each node N_i with a command c executes the following two steps to lock a position for c in a total ordering of commands:

1. N_i broadcasts $\langle \mathsf{RequestTS}, c \rangle_{\sigma_{N_i}}$ and waits for responses from $2f+1$ nodes, where σ_{N_i} is a signature on the payload using N_i's private key.

   - A node N_j responds with $\langle \mathsf{ResponseTS}, c, ts \rangle_{\sigma_{N_j}}$, where ts is a timestamp from N_j's local timer.

2. N_i broadcasts $\langle \mathsf{Sequence}, c, T \rangle_{\sigma_{N_i}}$, where T is a set of $2f+1$ responses received in the first step, and waits for responses from a quorum of $2f+1$ nodes.

   - A node N_j *accepts* the broadcast message and adds it to its localSequencedSet if the assigned timestamp of c is higher than localAcceptThresholdTS. If so, N_j responds with $\langle \mathsf{SequenceResponse}, \mathsf{ack}, h \rangle_{\sigma_{N_j}}$; otherwise, it responds with $\langle \mathsf{SequenceResponse}, \mathsf{nack}, h \rangle_{\sigma_{N_j}}$, where h is the cryptographic hash of the Sequence message.

The second step above is crucial to establish stable prefixes in the sequence of commands. Intuitively, it requires every correct node N_j to refuse sequencing commands if their timestamp may be lower than that of a stable command. Note that, during sufficiently long periods of synchrony (which are necessary for liveness), nodes can get their commands sequenced in just two round-trips—a lower latency than recent BFT protocols [18, 68]. However, sequenced commands are not yet suitable for execution until they become stable: only then they are guaranteed that commands with lower timestamps will not be sequenced.

Nodes *can* execute commands speculatively in their localSequencedSet, but they must wait for the consensus phase to finish before externalizing output and be ready to perform selective reexecution if their speculation is incorrect.

(2) Consensus phase. The principal goal of the consensus phase is to ensure that all correct nodes agree that a certain prefix of the total order constructed in the previous phase is now stable, meaning that the prefix is forever immutable.

To accomplish this, Pompē employs any standard leader-based BFT SMR protocol (e.g., [16, 31, 68]) that offers a primitive to agree on a value for each slot in a sequence of consensus slots. We generically refer to this protocol as Consensus. For simplicity, we assume that each consensus slot is associated with non-overlapping time intervals $[ts, ts')$ such that $ts' > ts$, and that for the first consensus slot $ts = 0$. We further assume that the mapping from consensus slot numbers to time intervals is common knowledge. In practice, this can be implemented by making the interval of the first consensus slot as $[0,\tau)$, where τ is the system initialization time, and then assigning each subsequent consensus slot a fixed window of time (e.g., $[ts, ts + 100\,\text{ms})$). Note that this does not mean that nodes must agree on a value during these time intervals.

For liveness, Pompē relies on a bound Δ on the sum of two terms: the maximum difference Δ_1 between the values returned, at any time, by local timers of correct processes, which in turn depends on the time it takes for a Sync to travel from one node to another and be processed at the recipient; and the maximum time Δ_2 needed by a correct node to execute the ordering phase (we assume that these bounds include additional slack to account for clock drifts across nodes). Pompē's safety properties hold even when Δ does not hold, but, during sufficiently long periods of synchrony (which is necessary for liveness), we assume that the bound holds for proving liveness (Section 4.2).

Local state. The local state of each node is a totally-ordered ledger, initially empty.

Actions. Suppose that consensus slot k maps to time inter-

val $[ts, ts')$, meaning that all commands with assigned timestamp in this interval are expected to be included in this slot. If node N_i wishes to serve as a leader in reaching consensus on a value for slot k using Consensus, it proceeds as follows.

1. N_i broadcasts $\langle \text{Collect}, k \rangle_{\sigma_{N_i}}$, and waits for responses from $2f+1$ nodes.
   - Node N_j waits until two conditions hold. First, the value of N_j's globalSyncTS is higher than ts', meaning that some node sent N_j a Sync message with $\mathcal{T} \geq ts'$. Second, since that Sync message was received, a time interval of at least Δ has elapsed on N_j's timer (i.e., N_j's timer reads at least localSyncTS $+\Delta$). Note that, during sufficiently long periods of synchrony, these delays give all correct nodes enough time to sequence all their commands with assigned timestamps lower than ts' before N_j advances its localAcceptThresholdTS to ts'. In more detail, after Δ_1, all correct nodes should have received and processed a Sync message with $\mathcal{T} \geq ts'$ to set their local timer to be at least $\mathcal{T}$, so after this point, any new command entering the ordering phase will not have an assigned timestamp lower than ts'. After an additional Δ_2, any command with an assigned timestamp lower than ts' must have completed the ordering phase.
   - N_j updates its localAcceptThresholdTS $\leftarrow max(ts', \text{localAcceptThresholdTS})$.
   - N_j responds with $\langle \text{CollectResponse}, k, \mathcal{S} \rangle_{\sigma_{N_j}}$, where $\mathcal{S}$ is the set of messages in the localSequencedSet of N_j with assigned timestamps in the interval $[ts, ts')$.
2. N_i runs Consensus to agree on value $\mathcal{U}$ for consensus slot k, where $\mathcal{U}$ is the union of CollectResponse messages from $2f+1$ nodes for consensus slot k.

Constructing a totally-ordered ledger. Once a prefix of consensus slots are agreed upon, nodes can construct a totally-ordered prefix of the ledger by sorting commands in each slot (of the prefix) by their assigned timestamps, breaking ties by their cryptographic hashes. When a node adds a proposal to its totally ordered ledger, it can execute them in the order specified by the ledger.

4.2 Proofs of safety and liveness

This section proves that Pompē satisfies ordering linearizability and a strengthened version of liveness in addition to standard safety properties.

Theorem 4.1 (Consistency)**.** *For every pair of correct nodes N_i and N_j with local ledgers $\mathcal{L}_i$ and $\mathcal{L}_j$, the following holds: $\mathcal{L}_i[k] = \mathcal{L}_j[k]\ \forall k :: 0 \leq k \leq \min(len(\mathcal{L}_i), len(\mathcal{L}_j))$, where $len(\cdot)$ computes the number of entries in a ledger.*

Proof. By the safety properties of BFT SMR, every pair of correct nodes agrees on the same value for each consensus slot. Furthermore, the transformation from values in consensus slots to a totally-ordered ledger is deterministic. Together, these observations imply the desired result. □

Theorem 4.2 (Validity)**.** *If a correct node appends a command c to its local totally-ordered ledger, then at least one node in the system proposed c in the ordering phase.*

Proof. Each command in the ledger of a correct node is constructed from a valid value agreed upon in one of the consensus slots. Furthermore, for a given consensus slot k with assigned time interval $[ts, ts')$, by our construction, a valid value is a set of CollectResponse messages for slot k from at least $2f+1$ nodes, where each CollectResponse contains commands with timestamps in the interval $[ts, ts')$. Additionally, for a command to have an assigned timestamp, it must have been proposed in the first step of the ordering phase. Together, these observations imply the statement of the theorem. □

Lemma 4.1. *The assigned timestamp of a command is bounded by timestamps provided by correct nodes.*

Proof. By assumption, there are at most f Byzantine nodes. Thus, at least $f+1$ (out of $2f+1$) timestamps provided in the ordering phase for a given command are from correct nodes. Furthermore, the assigned timestamp of a command discards f lowest and f highest timestamps in the $2f+1$ ResponseTS messages, thus the assigned timestamp of a command is bounded by timestamps provided by correct nodes. □

Theorem 4.3 (Ordering linearizability)**.** *If the highest timestamp provided by any correct node for a command c_1 is lower than the lowest timestamp provided by any correct node for another command c_2 and if both c_1 and c_2 are committed, then c_1 will appear before c_2 in the totally-ordered ledgers constructed by correct nodes.*

Proof. By Lemma 4.1, the assigned timestamp of a command is bounded by timestamps provided by correct nodes. As a result of this and the pre-condition in the statement of the theorem, the assigned timestamp of c_1 will be smaller than the assigned timestamp of c_2. Thus, if both c_1 and c_2 are committed, c_1 will appear before c_2 in the totally-ordered ledgers of correct nodes because nodes sort commands by their assigned timestamps. □

Lemma 4.2. *During sufficiently long periods of synchrony, a correct node can get its command (along with its assigned timestamp) added to localSequencedSet of at least $2f+1$ nodes.*

Proof (sketch). Suppose a correct node executes the first step of the ordering phase for its command c and obtains an assigned timestamp of ts. During sufficiently long periods of synchrony, by the choice of Δ, a Sequence message that includes c will reach $2f+1$ correct nodes and be added to their localSequencedSet *before* they advance their localAcceptThresholdTS past ts, which implies the statement of the lemma. □

Lemma 4.3. *If a command c with assigned timestamp ts is added to localSequencedSet of at least $2f+1$ nodes, then c will eventually be included in the value committed by a unique consensus slot whose time interval includes ts.*

Proof (sketch). Let k denote the consensus slot whose time interval includes *ts*. When a leader broadcasts Collect for consensus slot k, the local timers on correct nodes will eventually meet the condition required to send CollectResponse messages. Since c appears in the localSequencedSet of at least $2f+1$ nodes, and, by assumption, since at most f of them are Byzantine, at least $f+1$ nodes will include c in their CollectResponse for consensus slot k. Denote these $f+1$ nodes with $\mathcal{C}$.

Since Pompē's use of BFT SMR requires proposals that are constructed by taking a union of $2f+1$ CollectResponse messages, a leader must include at least one message from nodes in $\mathcal{C}$. Thus, c must be included to construct a valid proposal for consensus slot k. These combined with the liveness property of the employed BFT SMR protocol (which ensures that a valid value will eventually be chosen for each consensus slot) implies the desired result. □

Theorem 4.4 (Strong liveness)**.** *During sufficiently long periods of synchrony, a correct node can get an assigned timestamp for its command c such that c will eventually be included in the total order constructed by correct nodes at a position determined by the assigned timestamp of c.*

Proof. During sufficiently long periods of synchrony, by Lemmas 4.2 and 4.3, c will eventually be included in the value committed by a unique consensus slot whose time interval includes the assigned timestamp of c. Since the algorithm to construct a total ordering of commands from values committed by consensus slots sorts commands by their assigned timestamps, the position of c is determined by the assigned timestamp of c. □

4.3 Byzantine influence in Pompē

Pompē greatly diminishes the leverage of Byzantine nodes. Once a command is sequenced, Byzantine nodes can neither censor it nor affect its position in the totally-ordered ledgers of correct nodes. Furthermore, they cannot violate ordering linearizability. Nonetheless, as we saw in Sections 2 and 3, in a Byzantine democracy, it is impossible to completely eliminate the influence of Byzantine nodes, and Pompē is not immune from it.

Byzantine democracy in action. Consider the following execution of Pompē, where $n = 4$ and $f \leq 1$. There are two commands, c_1 and c_2, that in the ordering phase obtained the following timestamps from a quorum of $2f+1$ nodes.

	N_1	N_2	N_3
c_1	0	3	3
c_2	1	4	2

Assume, without loss of generality, that N_3 is Byzantine, and that the remaining nodes are correct. The timestamps make clear that correct nodes prefer to order c_1 before c_2. However, since the median timestamp of c_1 is higher than the median timestamp of c_2, it is c_2 that will be ordered before c_1. On a positive note, we observe that, in the normal case where the timers on correct nodes are sufficiently synchronized and network delays are small, this window of vulnerability to Byzantine manipulation is small.

Early stopping and deferred selective inclusion. Pompē cannot prevent a Byzantine node from obtaining an assigned timestamp for its command, but not proceeding with the rest of the ordering phase, as this misbehavior is indistinguishable from what may result from a network failure. This ambiguity allows a Byzantine node (possibly with the aid of a Byzantine leader) to decide later, during the consensus phase, whether or not to include its timestamped-but-not-yet-sequenced command in the ledger.

Preventing or reliably detecting this type of misbehavior is impossible, but mechanisms to mitigate the risks and raise suspicion do exist. One possibility is for each node to employ an append-only linear hash chain to record the timestamps it assigns to other nodes' commands. Nodes exchange those hash chains and refer to the corresponding hash value (in the hash chain) in each ResponseTS message. Such hash chains constrain the ability for Byzantine nodes to assign timestamps abnormally (e.g., out of order), and allow after-the-fact auditing (which could be used to expose nodes that routinely timestamp their commands, but do not always sequence those commands). In addition, a correct node N_i can piggyback the tail of a hash chain of all previously timestamped commands of N_j whenever N_j requests a timestamp; this makes it hard for a Byzantine N_j to blame on the network when silently dropping an earlier timestamped command. An alternative mechanism is for correct nodes to hide their commands using a threshold encryption scheme until those commands are totally ordered. This additional step prevents Byzantine nodes from observing the contents of other timestamped commands before deciding whether to drop their timestamped commands.

5 Implementation

We implement two variants of Pompē, where the artifacts differ in the specific BFT protocol they employ for the consensus phase. Specifically, we extend two prior state-of-the-art leader-based BFT protocols: SBFT [31] and HotStuff [68]. SBFT implements a variant of PBFT [16] that includes many optimizations for scalability. HotStuff uses a rotating leader paradigm while incurring low network costs and serves as the foundation of the Libra blockchain [3]. For SBFT, we use its implementation in VMware's Concord [1], and for HotStuff, we use the authors' implementation [2].

	base	extensions
Concord [1]	22,141	1122
HotStuff [68]	4,983	900

FIGURE 2—Number of lines of C++ code in Pompē, which we build atop a base BFT library with a set of extensions.

Ease of implementation. Implementing Pompē atop an existing consensus protocol involves modest system effort. Figure 2 reports the numbers of lines of code we add to the base BFT protocol implementations. These extensions primarily focus on implementing the two steps of the ordering phase in our new architecture. Specifically, we implement four new message types, as described in Section 4. We then implement message handlers to sign and verify timestamps and to manage data structures for localSequencedSet and localAcceptThresholdTS. Additionally, we modify the leader logic so that, for each time interval, a leader starts a consensus phase after assembling a proposal by collecting responses from a quorum of $2f + 1$ nodes, as described in Section 4. The rest of the consensus protocol is unmodified: the leader of an instance runs the original consensus protocol for a slot with a proposal assembled as described above. Within each slot, commands are ordered by their assigned timestamps.

Optimizations. In Pompē's consensus phase, the CollectResponse message used for consensus slot k contains all commands in a node's localSequencedSet whose assigned timestamp falls within the time interval associated with k. This can lead to large message sizes. However, when the network is synchronous and correct nodes respond in a timely manner, CollectResponse messages will contain the same set of commands. Therefore, we optimize Pompē by having CollectResponse messages sent to the leader carry only a hash of the set commands in the sender's localSequencedSet. The leader compares the hash of its own localSequencedSet with the hashes carried in the CollectResponse messages received from $2f$ other nodes. If the hashes match, then the leader proceeds to reach consensus for slot k on the commands from its localSequencedSet, using the $2f + 1$ signed hash values (those received from the other nodes as well as its own) as proof that $2f + 1$ nodes reported the same set of commands. Otherwise, the leader requests a new set of CollectResponse messages, this time including the actual set of commands. We enable this optimization by default.

6 Experimental evaluation

This section experimentally evaluates Pompē. We ask two main questions: (1) How does the performance of Pompē compare with that of state-of-the art BFT protocols? (or, what is the price of transitioning from a Byzantine oligarchy to a Byzantine democracy that enforces Byzantine-tolerant ordering guarantees?) and (2) What is the impact of separating ordering from consensus on end-to-end performance? Figure 3 provides a summary of our findings.

We choose as baselines two prior state-of-the-art BFT protocol implementations: Concord [1, 31] and HotStuff [2, 68]. Both are leader-based (and hence subject to Byzantine oligarchy) and hardcode ordering decisions within consensus. As described in Section 5, we implement two variants of Pompē, both upholding ordering linearizability (and hence free of Byzantine oligarchy), by augmenting those two BFT protocols. We refer to Pompē that extends HotStuff as *Pompē-HS*, and to Pompē that extends Concord as *Pompē-C*.

Methodology, testbed, and metrics. We run our experiments on 100 Standard D16s_v3 (16 vcpus, 64 GB memory) VMs on the Azure cloud platform spanning three datacenters, each running Ubuntu Linux 18.04: 34 in West US, 33 in South-East Asia, and 33 in North Europe. We run single-datacenter experiments using VMs in the West US.

We report results only for failure-free executions, as failures do not alter how Pompē performs relative to its baselines.

Our workload is generated by clients that submit their commands in a closed loop, i.e., they wait to receive a response to their currently outstanding command before submitting the next one. To run experiments with different loads, we vary the number of clients. For HotStuff and Pompē-HS, as in prior work [67], we run experiments where commands are random, 32-bytes-long values.[2]

Similarly, for Concord and Pompē-C, as in prior work [31], we use a benchmark that writes a random value to a randomly-selected key in a key-value store.

Our principal performance metrics are client-perceived latency (measured in ms) and throughput (in commands/second). To measure latency, each client records the latency of each command using its local clock, and our scripts aggregate latencies across clients and across commands. For throughput, we compute the total number of commands processed by the system and divide it by the duration of the experiment. To measure the peak throughput of a given system, we increase the number of clients until saturation.

Since Pompē separates ordering from consensus, clients in Pompē receive two responses, one for confirming the relative position of the command in the totally-ordered ledger (when a command is sequenced; see Section 4 for details), and another for the execution result of the command. Therefore, we report two types of latency for Pompē, which we refer to as *ordering latency* and *consensus latency*. Since our baselines hardcode ordering decisions within consensus, both ordering and consensus complete at the same time, so, for baselines, we report a single type of latency.

6.1 End-to-end performance: Throughput and latency

We begin by measuring the performance of Pompē and its baselines in a four-node configuration (we report results for

[2]The HotStuff implementation reaches consensus not on actual commands, but on their 32-byte-long cryptographic hashes; clients communicate the actual commands to the replica nodes outside of the consensus protocol.

Pompē incurs higher latency than its baselines, but by batching in both phases, Pompē achieves higher throughput at competative latencies	§6.1,6.2
Pompē's throughput degrades when n increases, but Pompē can scale up each node for higher throughput	§6.3
Pompē incurs modest network overheads over its baselines	§6.4

FIGURE 3—Summary of evaluation results.

	throughput (cmds/s)	median latency (ms)
HotStuff ($\beta_c = 1$)	474	8.2
HotStuff ($\beta_c = 800$)	253,360	49.9
Pompē-HS ($\beta_o = 1$)	1,642	2.3 (o), 47.7 (c)
Pompē-HS ($\beta_o = 200$)	361,687	5.7 (o), 53.1 (c)
Concord ($\beta_c = 1$)	40	53
Concord ($\beta_c = 800$)	6,633	67
Pompē-C ($\beta_o = 1$)	1,415	17 (o), 67 (c)
Pompē-C ($\beta_o = 200$)	249,221	18 (o), 74 (c)

FIGURE 4—Peak throughput and median latency for Pompē and its baselines in a single datacenter with $n = 4$ nodes. Pompē's leader starts the consensus phase every 50 ms with $\Delta = 10$ ms. Pompē's *ordering latency* is denoted with "o", its *consensus latency* with "c".

larger system sizes in the next subsection). We run clients on a separate set of virtual machines so that clients and nodes do not contend for computing resources.

A note about batching. Batching is a standard technique in SMR protocols to increase throughput at the cost of higher latency by amortizing the cost of running consensus across all the commands in a batch. Both Pompē and its baselines can take advantage of it, and we report experiments for different batch sizes. However, Pompē's separation of ordering from consensus has two significant implications for batching.

First, it eliminates the unintended leverage that Byzantine nodes can gain through batching even in BFT SMR protocols that rotate leaders out of concern for "fairness". The larger the batch, the larger the number of commands whose ordering is left to the unchecked discretion of the current leader: throughput gains thus come at the cost of expanding opportunities for Byzantine oligarchy. Pompē removes these concerns: its ordering guarantee (e.g., ordering linearizability) is unaffected by either the existence of batches or by their sizes.

Second, separating order and consensus affects the tradeoff between latency and throughput that comes with batching. When Pompē's baselines do not batch commands, they achieve lower latency *and* lower peak throughput than Pompē. Latency is higher under Pompē because a leader in Pompē must wait for a fixed time window before initiating a proposal; peak throughput is higher because Pompē implicitly batches commands whose timestamps fall within a time window during consensus. However, when the baselines batch commands to match Pompē's latencies, they achieve significantly higher peak throughput than Pompē. Pompē's peak throughput is lower because nodes must produce and validate signed timestamps during the ordering phase, which causes nodes to saturate earlier.

	throughput (cmds/s)	median latency (ms)
HotStuff ($\beta_c = 800$)	6,160	915.8
Pompē-HS ($\beta_o = 200$)	315,753	259.7 (o), 1518.1 (c)
Concord ($\beta_c = 800$)	1,461	616
Pompē-C ($\beta_o = 200$)	172,774	325 (o), 1415 (c)

FIGURE 5—Peak throughput and median latency for Pompē and its baselines with $n = 4$ nodes spanning three geo-distributed datacenters. Batch sizes are as in the single datacenter experiments in a single datacenter. Pompē's leader starts the consensus phase every 500 ms with $\Delta = 400$ ms.

Fortunately, the separation gives Pompē an additional batching opportunity: each node can execute the ordering phase once to assign a single timestamp to an ordered sequence of its own commands (or of commands from clients that belong to the same organization as the node). Such batching does not affect Pompē's ordering properties (e.g., ordering linearizability) because each batch contains commands from a single node. The throughput boost that comes from this additional source of batching can more than make up for Pompē's lost ground, but raises the question of how to fairly compare the Pompē variants to their baselines.

We balance these different considerations in our experiments as follows: if, in a configuration with n nodes, the baseline's consensus protocol uses a batch size $\beta_c = S(> 1)$, then we allow each node in corresponding variant of Pompē to use batches of size $\beta_o = S/n$ during its ordering phase.

Performance results. Figure 4 shows peak throughput and median latency at peak throughput for Pompē and its baselines, for different batch sizes. Since Pompē-C and Pompē-HS perform similarly compared with their respective baselines, so we focus only on Pompē-HS.

Performance without batching. When $\beta_o = 1$, Pompē-HS's median ordering latency is 28% of the median latency of HotStuff with $\beta_c = 1$, while its peak throughput is about 3.5× higher than HotStuff's. The lower ordering latency is due to Pompē's ordering phase, which incurs only two RTTs compared to the four RTTs required by HotStuff; the higher throughput, perhaps surprisingly given that $\beta_o = 1$, is instead due to batching. In Pompē-HS, setting $\beta_o = 1$ means that nodes do not batch in the *ordering phase*; however, since Pompē-HS does not start consensus until a time window has elapsed, it can still collect commands from multiple clients: for a 50 ms time window, we observed an effective batch size of 82 commands. Unsurprisingly, the flip side of this higher throughput is significantly higher consensus latency. Pompē-HS starts the consensus phase every 50 ms; with $\Delta = 10$ ms,

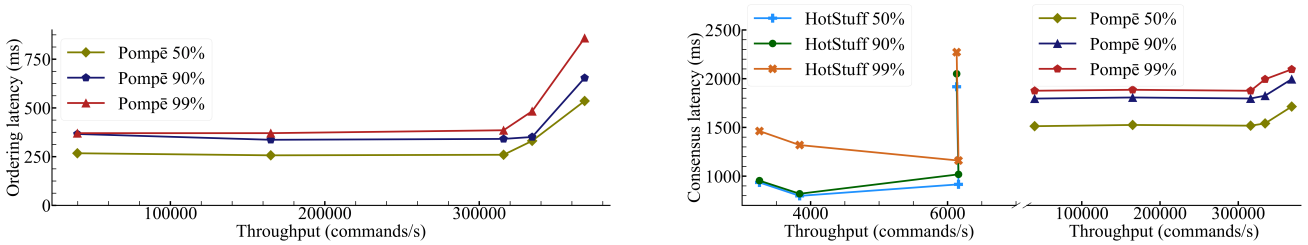


FIGURE 6—Latency vs. throughput for HotStuff and Pompē-HS in a geo-distributed deployment. The left and right graphs show respectively the maximum ordering latency and consensus latency experienced by different percentiles of the fastest commands. The experimental setup is the same as in Figure 5. Pompē-HS achieves higher throughput at the cost of higher consensus latency, even as its low ordering latency lets nodes know quickly when their commands are guaranteed to appear in the ledger.

every client waits on average 35 ms for the next consensus phase, ultimately leading to a consensus latency of 47.7 ms.

Performance with batching. We fix the batch size for HotStuff to $\beta_c = 800$ commands, and accordingly set the ordering-phase batch size of each of the four nodes in Pompē-HS to $\beta_o = 200$. Unsurprisingly, the throughput increases significantly for both Pompē-HS and HotStuff, respectively by 220× and 535× over the values we measured for Pompē-HS ($\beta_o = 1$) and HotStuff ($\beta_c = 1$): both systems are CPU-bound, and batching allows them to amortize the cost of cryptographic operations across all commands in a batch. In absolute terms, we find that Pompē-HS achieves 1.4× the throughput of HotStuff; as discussed earlier, the reason is the additional batching effect due to the 50 ms interval that in Pompē-HS separates successive invocations of consensus.

6.2 Performance with a geo-distributed setup

We consider next a geo-distributed setup, where $n = 4$ nodes are deployed in three separate datacenters, with one datacenter running two nodes. We use the same batch size as in the single datacenter setup (i.e., $\beta_c = 800$ for baselines and $\beta_o = 200$ for each node's ordering phase for the corresponding Pompē variants).

Peak throughput. Figure 5 shows our results. For HotStuff, geo-replication causes throughput to drop dramatically, to only 2.4% of its value for the same configuration in a single datacenter. For geo-distributed Pompē-HS instead the loss is much more contained: throughput is at 87.3% of its single-datacenter value. Two main factors explain these results. First, as in the single-datacenter case, Pompē-HS can take advantage of effective batching, now with a time interval between successive proposal of 500 ms and $\Delta = 400$ ms; second, HotStuff is hampered by its use of rotating leaders, as a new leader does not propose a new batch until after collecting enough votes for the previous leader's batch: in a geo-distributed setting, this delay can become significant and negatively affect throughput.

Latency. Figure 6 shows the maximum ordering and consensus latencies experienced by the fastest 50%, 90%, and 99% of commands. The key take-away is that Pompē-HS achieves higher throughput at the cost of higher consensus latencies. As expected, in Pompē-HS both types of latency stay stable until system saturation. HotStuff's latency drops at the beginning because, with more clients, it fills up a batch more quickly while also increasing the throughput. Furthermore, the ordering latency is lower than the median consensus latency (since the latter adds more communication rounds to the former) meaning that nodes can get early notification for when their commands are guaranteed to appear in the ledger.

6.3 Scalability

To understand how well Pompē scales to a larger number of nodes, we experiment with increasing values of n. We vary the number of nodes in an experiment from 4 to 100. Our results for Pompē-C (in comparison with its baseline Concord) are qualitatively similar to our results for Pompē-HS (in comparison with HotStuff), so we focus on Pompē-HS.

HotStuff uses the same batch size as before (i.e., $\beta_c = 800$). For Pompē-HS, we experiment with three configurations.

1. Light: We set $\beta_o = 800/n$ and allocate a single VM to each node regardless of n.
2. Scale-up: We set $\beta_o = 800/n$ and, as n increases, so does proportionally the number of VMs associated with each node to equal $\lfloor n/4 \rfloor$. So, for example, for $n = 4$, we use one VM per node; but when $n = 10$, each node uses two.
3. Fixed batch: We set $\beta_o = 200$ regardless of n.

Figures 7 and 8 depict throughput and latency achieved by Pompē and its baselines for different values of n.

Throughput. HotStuff scales well as n grows, whereas throughput quickly degrades under Pompē-HS (light). This is because batch sizes under Pompē-HS (light) are inversely proportional to n, so throughput degrades as n increases. This is confirmed by the scaling behavior of Pompē-HS (fixed batch) where $\beta_o = 200$ regardless of n. Of course, using a fixed β_o regardless of n may not be desirable.

Fortunately, we find that Pompē-HS (scale-up) can achieve a behavior similar to Pompē-HS (fixed batch) without having to use a fixed β_o. In Pompē-HS (scale up), each node uses multiple VMs to run the ordering phase, thereby avoiding

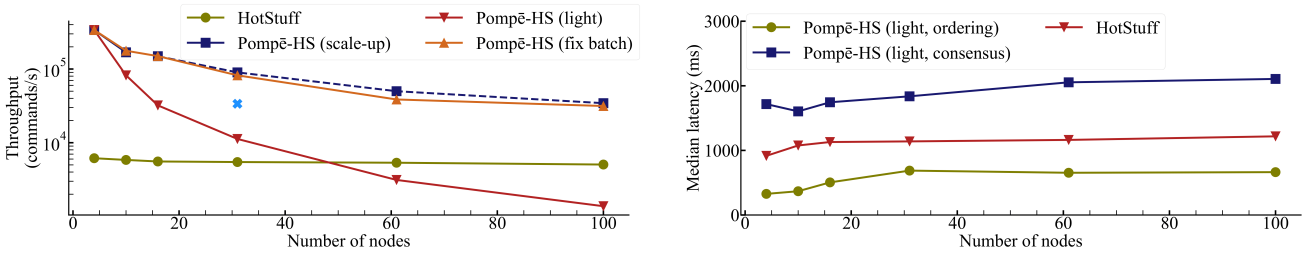


FIGURE 7—Peak throughput and median latency of different configurations of Pompē-HS and of HotStuff as a function of the number of nodes (n) in a geo-distributed deployment. The light blue cross at $n = 31$ depicts the performance of Pompē-HS (scale up) with 3 VMs per node; the blue square above it shows the *predicted* throughput when each node is assigned $\lfloor 31/4 \rfloor = 7$ VMs. The prediction is based on benchmarks showing that the ordering phase scales near linearly as more VMs are assigned to each node. The blue squares connected by a dotted line at $n = 61$ and $n = 100$ are similarly predicted rather than measured.

the throughput degradation experienced by Pompē-HS (light). Our testbed has 100 nodes, so we could only run Pompē-HS (scale-up) for $n \in \{4, 10, 16\}$. For higher values of n, we predict the throughput of Pompē-HS (scale-up) using experimental results from smaller-scale experiments and additional benchmarks that we used to validate that the ordering phase achieves a near-linear scaling as each node gets more VMs.

Latency. For both Pompē-HS and HotStuff, latency stays relatively stable when the system scales out. This is because latency is dominated by network communication in a geo-distributed deployment.

6.4 Network overhead

Compared to its baselines, Pompē incurs higher network costs to attach timestamps with each command and for executing a separate ordering phase. To understand the increased network costs, we use $n = 4$ and experiment with both Pompē and its baselines. We experiment with Pompē-HS ($\beta_o = 1$) and HotStuff ($\beta_c = 1$), and record the total number of bytes sent by each node during the experiment. We find that Pompē-HS incurs about 18% higher network costs compared to HotStuff, which, we believe, is a tolerable price for the stronger ordering properties ensured by Pompē.

7 Related work

Leader-based BFT protocols. There is a long line of work on practical Byzantine consensus protocols [10, 17, 20, 31, 34, 41, 42, 49–52, 59, 65, 66], starting with the seminal work of PBFT [16]. These works focus on improving performance, round complexity, fault models, etc. Some works also focus on using trusted hardware to improve fault thresholds [10, 19, 37, 46]. However, all of them employ a special leader node to orchestrate both ordering and consensus, so they suffer from both Byzantine dictatorship and Byzantine oligarchy.

There are some works that defend against faulty leaders, but they focus only on preventing faulty leaders from affecting the system's performance or defenses for a restricted class of attacks. For example, Aardvark [21] employs periodic leader changes to prevent a faulty leader from exercising full control over the system's performance. It achieves this by having correct nodes set an expectation on minimal acceptable throughput that a leader must ensure and trigger a leader election in case the current leader fails to meet its expectation. While Aardvark [21] focuses on achieving acceptable performance in the presence of faulty leaders, Prime [5] targets a different performance property: any transaction known to a correct node is executed in a timely manner. The Prime Ordering protocol consists of a pre-ordering phase and a global ordering phase. Unlike Pompē's ordering phase, the pre-ordering phase imposes only a partial order, rather than a timestamp-based global ordering in Pompē.

Instead of monitoring leaders to detect (or prevent) certain attack vectors, Pompē separates ordering from consensus, which completely eliminates a leader's power in selecting which transactions to propose and in what order. More generally, our work provides the first systematic study of properties desirable when employing BFT protocols for systems that span multiple administrative domains, proves what are impossible, and designs mechanisms to realize desirable properties that are achievable.

Rotating leaders. BART [4] enables cooperative services to tolerate both Byzantine faults and rational (selfish) behavior under the new BAR (Byzantine, altruistic, and rational) model. The consideration of rational behavior leads to an RSM design with rotating leaders, which has now become a standard practice for blockchains based on BFT [3, 18, 68]. However, the rotating leader paradigm still suffers from Byzantine dictatorship because a Byzantine node can still dictate ordering when it is in the leadership role, whereas Pompē achieves stronger properties by separating ordering from consensus.

Leaderless BFT protocols. Recognizing the implications of relying on a special leader, Lamport offers a leaderless Byzantine Paxos protocol [44]. Unfortunately, it relies on a synchronous consensus protocol to instantiate a "virtual" leader, which requires at least $f + 1$ rounds, where f is the maximum number of faulty nodes in the system and the duration of each round must be set to an acceptable round trip delays. When the number of nodes is high or when nodes

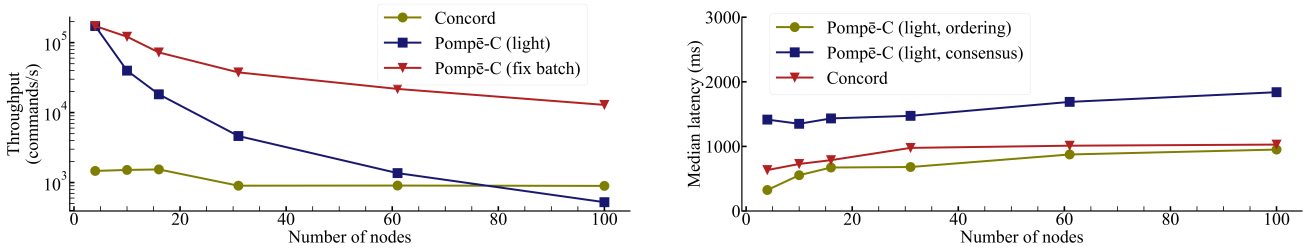


FIGURE 8—Scalability of Pompē-C and Concord in a geo-distributed deployment. Peak throughput and median latency with varying number of nodes (n). We use $\beta_c = 800$ for the baseline; see the text for different configurations of Pompē.

are geo-distributed, this protocol adds unacceptable latencies. Democratic Byzantine Fault Tolerance (DBFT) [23] is another leaderless Byzantine consensus protocol, which builds on Psync, a binary Byzantine consensus algorithm. As in Lamport's leaderless protocol [44], Psync terminates in $O(f)$ message delays, where f is the number of Byzantine faulty nodes, even though DBFT relies on a weak coordinator for a fast path through optimistic execution.

EPaxos [55] is a Paxos [43] variant in which proposed transactions are ordered without relying on a single leader. But EPaxos ensures safety and liveness only in a crash fault model, and it is unclear how to ensure those properties in a Byzantine fault model, which is our target setting.

Building on the work of Cachin et al. [14, 15], HoneybadgerBFT [54] and BEAT [26] propose leaderless protocols that preserve liveness even in asynchronous and adversarial network conditions. To achieve these properties, they rely on randomized agreement protocols, which bring significant complexity and costs. Unfortunately, these works do not defend against the formation of a Byzantine oligarchy nor do they satisfy ordering linearizability.

Censorship-resistance. HoneybadgerBFT [54] and Helix [8] run consensus on transactions encrypted with a threshold encryption scheme to prevent malicious nodes from censoring transactions, but faulty nodes can always filter transactions based on metadata, a point made by Herlihy and Moir [35]. In contrast, Pompē's separation of ordering from consensus offers a simple mechanism to prevent censorship: once a correct node executes the ordering phase, the transaction is not only guaranteed to be included in the ledgers of correct nodes, it will also be included in a position determined by the assigned timestamp of the transaction.

Accountability and proofs. Herlihy and Moir [35] propose several mechanisms to hold participants accountable in a consortium blockchain. These techniques extend and generalize prior work on accountability [32, 33] and untrusted storage [48, 53]. Similarly, nodes can produce succinct (zero-knowledge) proofs of their correct operation, which other nodes can efficiently verify [12, 58, 63, 64]. Recent work [45, 56] employs such proofs to reduce CPU and network costs in large-scale replicated systems (e.g., blockchains). Unfortunately, such proofs do not prevent a Byzantine leader node from deciding which commands to propose and in what order.

Order fairness. Recent work by Kelkar et al. [38] also recognizes the need to introduce a new ordering property for BFT, which they characterize as order fairness. Their work shows that a natural definition of Receive-Order-Fairness, which states that the total order of commands in the consensus output must follow the actual receiving order of at least a γ-fraction of all nodes (if they agree), is impossible to achieve, due to the Condorcet paradox. They relax Receive-Order-Fairness and define Block-Order-Fairness, where ordering constraints apply only to *blocks* of commands.

Starting from a similar motivation, our work takes a different direction, with both theoretical and practical implications.

First, rather than trying to characterize the fairness of a particular ordering, we introduce the notions of Byzantine oligarchy and Byzantine democracy to focus on the degree to which it is possible (and impossible) to curtail the influence of Byzantine nodes in determining any given order of commands. Thus, while Kelkar et al. observe that protocols that order commands using timestamps from a quorum of nodes are not suitable for ensuring fairness (as they suffer from the type of manipulations described in Section 4.3), we are able to prove (see Theorem 3.1) that *any* protocol is subject to these types of manipulations in a Byzantine democracy, as long as we uphold free will.

Further, we choose to express our ordering properties as a function of the preferences of correct nodes, rather than some γ-fraction of all the nodes (some of which could be Byzantine); we believe this choice was instrumental in deriving clean definitions for ordering unanimity and ordering linearizability.

Our different design choices have also significant practical consequences. While Pompē can use any existing BFT protocol in its consensus phase, Kelkar et al. design a compiler to automatically convert a standard consensus protocol into one that satisfies order fairness. However, protocols output by this compiler require more resources than a standard BFT protocol for the same level of fault tolerance; for example, in the same setting as in standard BFT (leader-based, partial synchrony network model) with γ set to 1 (their best case), these protocols require at least $4f + 1$ nodes to tolerate f Byzan-

tine failures, rather than the $3f + 1$ nodes needed by Pompē. Further, the practicality of these compiler-produced protocols is unclear, since to date they appear to have been neither implemented nor evaluated, whereas Pompē is competitive with state-of-the-art BFT protocol implementations.

Permissionless blockchains. A trend in the blockchain community is to avoid energy-intensive proof-of-work mechanism. This has led to permissionless blockchains that employ a BFT protocol among a set of nodes chosen based on different mechanisms (e.g., verifiable random functions, financial stake, etc.) to agree on a value [25, 30, 39, 40]. Pompē can be used as a building block in some of these blockchains.

Social choice theory. Social choice theory studies desirable properties in the context of elections. A seminal work in this area is by Kenneth Arrow [7], who won the Nobel Prize in Economics Sciences in 1972 for this work. Arrow's work defines properties such as *non-dictatorship* and *unanimity*, which inspired our definitions of Byzantine oligarchy and ordering unanimity. Following Arrow's work, Gibbard and Satterthwaite defined the *manipulation* property and proved that any voting rule is either dictatorial or manipulable [29, 61]. This property inspired our definition of Byzantine democracy. Finally, in the past two decades, computer scientists became interested in social choice theory, leading to the creation of the field of computational social choice [11].

8 Discussion

Deployment models. Section 4 describes our protocol in a simplified deployment model centered on nodes, without explicitly mentioning clients, for ease of exposition. This is a reasonable model in the context of our target application of permissioned blockchains, where each node is owned and operated by a separate organization: we can expect clients that belong to an organization to submit their transactions to a node owned by the same organization (so the incentives of clients and nodes are aligned). This deployment model also increases the opportunity for batching in the ordering phase at each node on behalf of all clients in the same organization.

Nevertheless, other deployment models are possible (e.g., those involving clients explicitly without associating them with trusted organizational nodes). Pompē's separation of ordering from consensus makes the following possible: each client executes the ordering phase with nodes for its commands and nodes execute the consensus phase. The protocol does have to account for the revised client/node communication pattern in the calculation of the delay (previously, Δ) in the consensus phase to ensure liveness, as well as handling duplicate requests from clients to different nodes to ensure that one of the nodes is correct and will accept the request.

Powerful network adversaries. Our network model assumes partial synchrony (as do prior BFT protocols). This does not eliminate a network-level adversary from affecting the assigned timestamps of commands. For example, a powerful adversary that controls the entire network connecting honest nodes can selectively reorder or delay messages among honest nodes to bias timestamps assigned to commands. Unfortunately, it appears impossible to completely curb the influence of such powerful network adversaries.

Another commonly adopted network-adversary model [51] assumes that an adversary cannot influence the network connecting correct nodes. In this model, an adversary does not gain additional power in biasing the assigned timestamps beyond what Byzantine nodes could already do.

Command dependencies or replay protection. As in prior BFT protocols, Pompē does not consider dependencies among different commands, nor does it prevent the same command from appearing multiple times in the total order. However, one can embed additional metadata inside commands (e.g., nonces, explicit dependencies, etc.), which correct nodes can use at the time of execution (i.e., after Pompē's consensus phase outputs a total order) to enforce dependencies among commands or to defend against replay attacks.

9 Concluding remarks

Pompē is a new, practical, and surprisingly simple BFT protocol that demonstrates an ideal world of Byzantine democracy, where free will is respected, under the "constitution" of ordering linearizability, and is not subject to Byzantine oligarchy. And this ideal world has been shown to operate competitively against the traditional world with Byzantine dictatorship.

Pompē's source code along with instructions to reproduce our experimental results will be available from: `https://github.com/pompe-org`.

Acknowledgments

We thank Frans Kaashoek (our shepherd) and the anonymous OSDI reviewers for their thorough and insightful comments. Trevor Eberl, Jim Jernigan, and Kris Zentner offered timely help with setting up a large-scale cluster on Azure. The initial steps towards a theory of Byzantine ordered consensus benefited from early conversations with Florian Suri-Payer and Mahimna Kelkar, and the help of Maofan Yin was invaluable in making it possible to use HotStuff as one of our baselines. This work was supported in part by NSF grants CSR-17620155 and CNS-CORE 2008667.

References

[1] Concord Byzantine fault tolerant state machine replication library. `https://github.com/vmware/concord-bft`, 2018.

[2] libhotstuff: A general-purpose BFT state machine replication library with modularity and simplicity. `https://github.com/hot-stuff/libhotstuff`, 2018.

[3] State machine replication in the Libra blockchain. `https://developers.libra.org/docs/state-machine-replication-paper`, 2020.

[4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In

Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), pages 45–58, 2005.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, July 2011.

[6] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2018.

[7] K. J. Arrow. *Social choice and individual values*, volume 12. Yale University Press, 1951.

[8] A. Asayag, G. Cohen, I. Grayevsky, M. Leshkowitz, O. Rottenstreich, R. Tamari, and D. Yakira. A fair consensus protocol for transaction ordering. In *Proceedings of the International Conference on Network Protocols (ICNP)*, 2018.

[9] D. Austen-Smith and J. S. Banks. *Positive political theory I: Collective preference*, volume 1. University of Michigan Press, 2000.

[10] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[11] F. Brandt, V. Conitzer, U. Endriss, J. Lang, and A. D. Procaccia. *Handbook of computational social choice*. Cambridge University Press, 2016.

[12] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[13] E. Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. Master's thesis, The University of Guelph, 2016.

[14] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 524–541, 2001.

[15] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings of the Internal Conference on Dependable Systems and Networks (DSN)*, pages 167–176, 2002.

[16] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[17] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, pages 236–269, 2003.

[18] B. Y. Chan and E. Shi. Streamlet: Textbook streamlined blockchains. Cryptology ePrint Archive, Report 2020/088, 2020.

[19] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–204, 2007.

[20] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2009.

[21] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

[22] M. d. Condorcet. Essay on the application of analysis to the probability of majority decisions. *Paris: Imprimerie Royale*, 1785.

[23] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. DBFT: Efficient leaderless byzantine consensus and its application to blockchains. In *Proceedings of the International Symposium on Network Computing and Applications (NCA)*, 2018.

[24] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[25] P. Daian, R. Pass, and E. Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Proceedings of the International Financial Cryptography Conference*, 2019.

[26] S. Duan, M. K. Reiter, and H. Zhang. BEAT: Asynchronous BFT made practical. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 2028–2041, 2018.

[27] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2), Apr. 1988.

[28] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the Symposium on Principles of Database Systems*, pages 1–7, 1983.

[29] A. Gibbard. Manipulation of voting schemes: a general result. *Econometrica: Journal of the Econometric Society*, pages 587–601, 1973.

[30] Y. Gilad, R. Hemo, S. M. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[31] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. SBFT: A scalable decentralized trust infrastructure for blockchains. arxiv:1804/01626v1, Apr. 2018.

[32] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for Byzantine fault detection. In *Proceedings of the USENIX Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[33] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188, 2007.

[34] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter. Zzyzx: Scalable fault tolerance through Byzantine locking. In *Proceedings of the Internal Conference on Dependable Systems and Networks (DSN)*, pages 363–372, 2010.

[35] M. Herlihy and M. Moir. Enhancing accountability and trust in distributed ledgers. *CoRR*, abs/1606.07490, 2016.

[36] M. P. Herlihy and J. M. Wing. Linearizability: A correctness

condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), July 1990.

[37] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 295–308, 2012.

[38] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels. Order-fairness for Byzantine consensus. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[39] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2017.

[40] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the USENIX Security Symposium*, 2016.

[41] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2007.

[42] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the Internal Conference on Dependable Systems and Networks (DSN)*, pages 575–584, 2004.

[43] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[44] L. Lamport. Leaderless Byzantine Paxos. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 141–142, Dec. 2011.

[45] J. Lee, K. Nikitin, and S. Setty. Replicated state machines without replicated execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[46] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2009.

[47] M. Lewis. *Flash boys: A Wall Street revolt*. W. W. Norton & Company, 2014.

[48] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[49] J. Li and D. Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[50] J. Liu, W. Li, G. O. Karame, and N. Asokan. Scalable Byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1), 2019.

[51] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic. XFT: practical fault tolerance beyond crashes. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 485–500, 2016.

[52] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[53] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1988.

[54] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT Protocols. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[55] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[56] A. Ozdemir, R. S. Wahby, and D. Boneh. Scaling verifiable computation using efficient set accumulators. In *Proceedings of the USENIX Security Symposium*, 2020.

[57] D. C. Parkes, C. Thorpe, and W. Li. Achieving trust without disclosure: Dark pools and a role for secrecy-preserving verification. In *Proceedings of the Conference on Auctions, Market Mechanisms and Their Applications (AMMA)*, 2015.

[58] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[59] D. Porto, J. a. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth fault tolerance. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 8:1–8:14, 2015.

[60] M. Russinovich, E. Ashton, C. Avanessians, M. Castro, A. Chamayou, S. Clebsch, M. Costa, C. Fournet, M. Kerner, S. Krishna, et al. CCF: A framework for building confidential verifiable replicated services. Technical report, Microsoft Research Technical Report MSR-TR-2019-16, 2019.

[61] M. A. Satterthwaite. Strategy-proofness and arrow's conditions: Existence and correspondence theorems for voting procedures and social welfare functions. *Journal of Economic Theory*, 10(2):187–217, 1975.

[62] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[63] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2018.

[64] S. Setty, S. Angel, and J. Lee. Verifiable state machines: Proofs that untrusted services operate correctly. *ACM SIGOPS Operating Systems Review*, 54(1):40–46, Aug. 2020.

[65] J. Sousa, A. Bessani, and M. Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 51–58. IEEE, 2018.

[66] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 253–267, 2003.

[67] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus in the lens of blockchain. *CoRR*, abs/1803.05069, 2018.

[68] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham.

HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes

Florian Rommel [1], *Christian Dietrich* [1], *Daniel Friesel* [2], *Marcel Köppen* [2], *Christoph Borchert* [2], *Michael Müller* [2], *Olaf Spinczyk* [2], *and Daniel Lohmann* [1]

[1] Leibniz Universität Hannover [2] Universität Osnabrück

Abstract

Live patching has become a common technique to keep long-running system services secure and up-to-date without causing downtimes during patch application. However, to safely apply a patch, existing live-update methods require the entire process to enter a state of quiescence, which can be highly disruptive for multi-threaded programs: Having to halt all threads (e.g., at a global barrier) for patching not only hampers quality of service, but can also be tremendously difficult to implement correctly without causing deadlocks or other synchronization issues.

In this paper, we present WFPATCH, a wait-free approach to inject code changes into running multi-threaded programs. Instead of having to stop the world before applying a patch, WFPATCH can gradually apply it to each thread individually at a local point of quiescence, while all other threads can make uninterrupted progress.

We have implemented WFPATCH as a kernel service and user-space library for Linux 5.1 and evaluated it with OpenLDAP, Apache, Memcached, Samba, Node.js, and MariaDB on Debian 10 ("buster"). In total, we successfully applied 33 different binary patches into running programs while they were actively servicing requests; 15 patches had a CVE number or were other critical updates. Applying a patch with WFPATCH did not lead to any noticeable increase in request latencies – even under high load – while applying the same patch after reaching global quiescence increases tail latencies by a factor of up to 41× for MariaDB.

1 Introduction

The internet has become a hostile place for always-online systems: Whenever a new vulnerability is disclosed, the respective fixes need to be applied as quickly as possible to prevent the danger of a successful attack. However, it is not viable for all systems to just restart them whenever a patch becomes available, as the update-induced downtimes become too expensive. The prime example for this are operating-system updates, where rebooting can take minutes. However, we increasingly see similar issues with system services at the application level: For example, if we want to update and restart an in-memory database, like SAP HANA or, at smaller scale, an instance of *Memcached* [11] or *Redis* [32], we either have to persist and reload their large volatile state or we will provoke a warm-up phase with decreased performance [26]. With the advent of nonvolatile memory [24], these issues will become even more widespread as process lifetimes increase [19] and eventually even span OS reboots [35]. In general, downtimes pose a threat to the service-level agreement as they provoke request rerouting and increase the long-tail latency.

A possible solution to the update–restart problem is dynamic software updating through live patching, where the patch is directly applied, in binary form, into the address space of the running process. However, live patching can also cause unacceptable service disruptions, as it commonly requires the entire process to become quiescent: Before applying the patch, we have to ensure that a safe state is reached (e.g., no call frame of the patched function f exists on any call stack during patching), which usually involves a global barrier over all threads – with long and potentially unbounded blocking time. In programs with inter-thread dependencies it is, moreover, tremendously difficult to implement such a barrier without risking deadlocks. To circumvent this, some approaches (such as UpStare [22]) also allow patching active functions, which involves expensive state transformation during patch application. Others (like KSplice [3]) probe actively until the system is in a safe state, which, however, is unbounded and may never be reached. Moreover, even in these cases it is necessary to halt all threads during the patch application. DynAMOS [23] and kGraft [29] avoid this at the cost of additional indirection handlers, but are currently restricted to the kernel itself as they rely on supervisor mechanisms. So, while disruption-free OS live patching is already available, live patching of multi-threaded user-space servers with potentially hundreds of threads is still an unsolved problem.

In a Nutshell We present WFPATCH, a wait-free live patching mechanism for multi-threaded programs. The fundamen-

tal difference of WFPATCH is that we do not depend on a safe state of *global quiescence* (which may never be reached) before applying a patch to the whole process, but instead can gradually apply it to each thread at a thread-specific point of *local quiescence*. Thereby, (1) no thread is ever halted, (2) a single hanging thread cannot delay or even prevent patching of all other threads, and (3) the implementation is simplified as quiescence becomes a (composable) property of the individual thread instead of their full orchestration. Technically, we install the patch in the background into an additional *address space (AS)*. This AS remains in the same process and shares all memory except for the regions affected by the patch – which then is applied by switching a thread's AS.

A current limitation of WFPATCH is that we can only patch read-only regions (.text and .rodata). In particular, we cannot apply patches that change the layout of data structures or global variables. However, WFPATCH is intended for hot patching and not for arbitrary software updates and the vast majority of software fixes are .text-only: In our evaluation with OpenLDAP, Apache, Memcached, Samba, Node.js, and MariaDB, this holds for 90 out of 104 patches (87%). For CVE mitigations and other critical issues, it holds for 36 out of 41 patches (88%).

This paper makes the following contributions:

- We analyze the qualitative and quantitative aspects of global quiescence for hot patching and suggest local quiescence as an alternative (Section 2, Section 4).
- We present the WFPATCH wait-free code-injection approach for multi-threaded applications and its implementation for Linux (Section 3).
- We demonstrate and evaluate the applicability of WFPATCH with six multi-threaded server programs (OpenLDAP, Apache, Memcached, Samba, Node.js, and MariaDB), to which we apply patches under heavy load (Section 4).

The patching procedure itself is out of scope for this paper, specifically, how binary patches are generated and what kind of transformations take place when applying them to an AS. Without loss of generality, we used a slightly modified version of Kpatch [30] to generate the binary patches for this paper. However, WFPATCH is mostly transparent in this regard and could be combined with any patch generation framework. We discuss its general applicability, the soundness and limitations and other properties of WFPATCH in Section 5 and related work in Section 6 before we conclude the paper in Section 7.

2 Problem Analysis: Quiescence

Most live-patching methods require the whole system to be in a safe state before the binary patch gets applied. Thereby, situations are avoided where the process still holds a reference to memory that is modified by the update. For example, for a

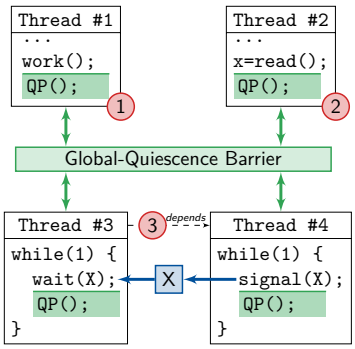


Figure 1: Problems of Global Quiescence. As all threads have to synchronize at the global-quiescence barrier, problems in individual threads can prolong the transition phase: (1) Long-running computations introduce bounded delays, (2) I/O wait leads to (potentially) unbounded barrier-wait times, and (3) inter-thread dependencies force a specific arrival order to avoid deadlocks.

patch that replaces a function f, the system is in a safe state if no call frame for f exists on the execution stack (denoted as *activation safety* in the literature [16]). Otherwise, it could happen that a child of f returns to a now-altered code segment and provokes a crash. While defining and reaching safe states is relatively easy for single-threaded programs, it is much harder for multi-threaded programs, like operating systems or network services.

In general, a safe state of a running process is a predicate Ψ_{proc} over its dynamic state S. For a multi-threaded process, we can decompose this predicate into multiple predicates, one per thread (th1, th2, ...), and the whole process is patchable iff all of its threads are patchable at the same time:

$$\Psi_{\text{proc}}(S) \Leftrightarrow \Psi_{\text{th1}}(S) \wedge \Psi_{\text{th2}}(S) \ldots$$

One possibility to bring a process into the safe state is to use *global quiescence* and insert *quiescence points* into the control flow: When a thread visits a quiescence point its Ψ_{thN} is true and we let the thread block at a barrier to keep the thread in this patchable state. One after another, all threads visit a quiescence point, get blocked at the barrier, and we eventually reach Ψ_{proc} after all threads have arrived. In this stopped world, we can apply all kinds of code patching and object translations [17, 15] as we have a consistent view on the memory.

However, *global quiescence* is problematic as it can take – depending on the system's complexity – a long or even *unbounded* amount of time to reach. Furthermore, eager blocking at quiescence points can result in deadlocks: If the progress of thread A depends on the progress of thread B, thread B must pass by its quiescence points until thread A has reached $\Psi_{\text{A}}(S)$. Even worse, in an arbitrary program, it is possible that $\Psi_{\text{C}}(S)$ and $\Psi_{\text{D}}(S)$ contradict each other such

that $\Psi_{\text{proc}}(S)$ can never be reached. Therefore, programmers need an in-depth understanding of the system to apply global quiescence without introducing deadlocks, and they must take special precautions to ensure that it is reachable eventually.

Figure 1 illustrates these problems. For example, if any thread in the system is performing a long-running computation when the patch request arrives, that is, *Problem 1*, the others will reach the barrier, which is now activated, one by one and stop doing useful work. During this transition-period clients will notice significant delays in response times and requests will queue-up or even time out. We have seen this problem in most of the systems that we examined. For example, Node.js threads perform long-running just-in-time compilation of Javascript code.

Similarly, in *Problem 2*, a thread is waiting on an IO operation. During this potentially unbounded period, other threads will reach the barrier. Again, the overall progress rate deteriorates before it becomes zero during the patching itself. This happens, for instance, when the Apache web server is transferring huge files to a client or executing a long-running PHP script. In an extreme case, the system could even have a thread that is waiting for interactive user input that never comes. Both problems are hard to avoid without changing the complete software structure by the programmer who has to insert quiescence points. Sometimes I/O operations can be quiescence points, but this is application-specific; for example, an I/O operation deep in the call stack or with locks held would be no suitable point for quiescence.

Problem 3 is more subtle and related to inter-thread dependencies. In MariaDB, for instance, worker threads perform database transactions and, thus, have to be synchronized. If a thread that is holding a lock reaches the barrier and blocks, a deadlock will occur if another thread tries to acquire that lock. In this case, the second thread would block and never reach the barrier to free the lock-holding thread. Therefore, a lock-holding thread must not enter the barrier, although its $\Psi_{\text{thN}}(S)$ is true, to avoid the cyclic-wait situation between barrier and lock. More generally speaking, applying global quiescence correctly requires full knowledge about *all* inter-thread dependencies where one thread's progress depends on another thread's progress.

In this paper, we mitigate the aforementioned problems by proposing the concept of local quiescence. Our main contribution is the concept of address-space generations, that is, slightly differing views of an AS that can be assigned on a per-thread basis. This makes it possible to prepare a patch in the background in a new AS and to migrate threads one-by-one to the patched universe. A global barrier is not needed. The approach is "wait-free" in the sense that a thread that has reached a quiescence point ($\Psi_{\text{th}N}(S)$ is true) can be patched immediately. Sections 4.2 and 5 discuss how this approach and its limitations apply to widely-used software projects.

Figure 2 illustrates the difference between the normal "global quiescence" approach (upper half) and the proposed

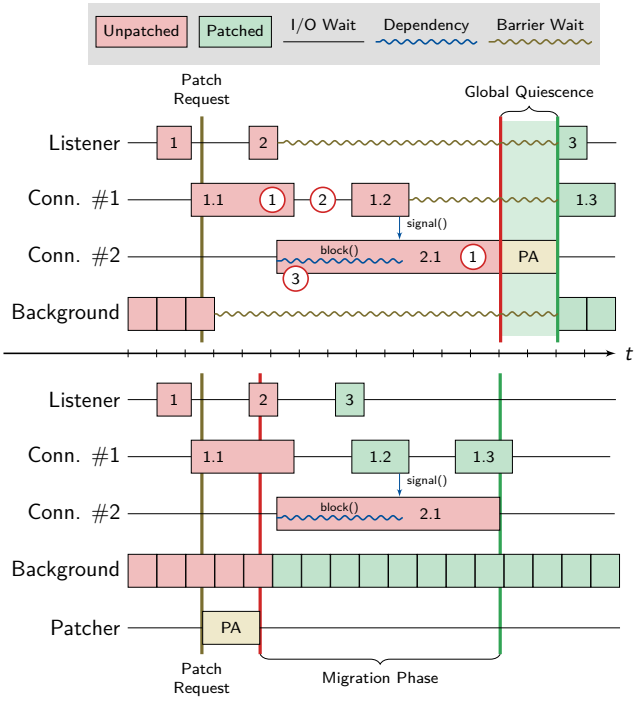


Figure 2: Live Patching a Multi-Threaded Server with Global (upper half) vs. Local (lower half) Quiescence. The global quiescence approach suffers from Problem 1–3 (see Figure 1) while the threads with the local quiescence model can be migrated to the patched state individually.

"local quiescence" (lower half). The scenario is a database server with a "Listener" thread for accepting connections, connection threads ("Conn. #1 and #2") for each client connection, and a "Background" thread for cleanup activities. The patch request comes in asynchronously while the listener is accepting the second connection. At this point in time "Conn. #1" has already started a transaction and is holding a lock. In the upper half (global quiescence) we find all three problems again. For example, the computation time of 1.1 and 2.1 as well as the I/O wait between 1.1 and 1.2 delay the patch application. During this period, the listener does not accept any new connections (request 3) and the background thread is blocked. Furthermore, the programmer must make sure that "Conn. #1" does not block at the barrier before executing 1.2 and releasing the transaction lock, as this would lead the whole system into a deadlock. With local quiescence, each thread can be migrated to the patched program version individually. Thus, no artificial delays are introduced and the quality of service is unaffected. For all but one thread the patch is applied earlier than in the global quiescence case. These seconds might be crucial in the case of an active security attack. Furthermore, deadlocks cannot occur as long as the patched version of the code releases the transaction lock.

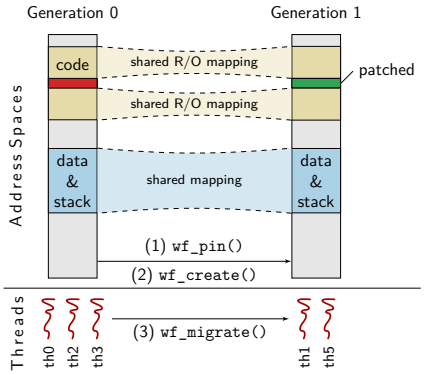


Figure 3: Process during the Wait-Free Patching

3 The WFPATCH Approach

Most previous live-patching mechanisms require a global safe state before applying the changes to the *address space (AS)* of the process. With our approach (see Figure 3), we reverse and weaken this precondition with the help of the OS and a user-space library: Instead of modifying the currently-used AS, we create a (shallow) clone AS inside the same process, apply the modifications there in the background, and migrate one thread at a time to the new AS, whenever they reach a local quiescence point, where their Ψ_{thN} becomes true. In the migration phase, we require no barrier synchronization and all threads make continuous progress. After the migration is complete, we can safely drop the old AS.

While both AS generations exist, we synchronize memory changes efficiently by sharing all unmodified mappings between old AS (Generation 0) and the new AS (Generation 1): We duplicate the *memory-management unit (MMU)* configuration but reference the same physical pages. Thereby, all memory writes are instantaneously visible in both ASs and even atomic instructions work as expected. Only for patch-affected pages, we untie the sharing lazily with existing *copy on write (COW)* mechanisms.

3.1 System Interface

As WFPATCH requires a kernel extension for handling multiple AS generations per process, we introduce four new system calls: `wf_create()`, `wf_delete()`, `wf_pin()`, and `wf_migrate()`. By the integration into the kernel, we are able to modify the AS without halting the whole process.

With `wf_create()`, the kernel instantiates a new AS generation which is a clone of the process's current AS. Any thread, even from a signal handler, can invoke `wf_create()`. AS generations are identified by a numeric ID and can be deleted with the `wf_delete()` system call. We keep AS generations in sync and changes to the AS are equally performed on all generations.

With `wf_pin()`, we can configure, in advance, memory regions that are not shared between AS generations. Within pinned regions, memory writes and page-protection changes will only affect the AS generation of the current thread. Thereby, we are able to have AS generations that differ only in patched pages.

On creation, new AS generations host no threads, but individual threads migrate explicitly by calling `wf_migrate(AS)`. On migration, the kernel modifies the *thread control block (TCB)* to use the patched AS, and the thread continues immediately once the system-call returns. For live patching, threads invoke `wf_migrate()`, via our user-space library, at their local-quiescence points.

3.2 Implementation for Linux

We implemented the WFPATCH kernel extension as a patch with 2000 (added or changed) lines for Linux 5.1. We tested and evaluated WFPATCH on the AMD64 architecture but it should work on every MMU-capable architecture supported by Linux. The basic idea is to clone address spaces in a fork-like manner and rely mostly on the page-sharing mechanism to keep clones lightweight and efficient. In contrast to fork, we do not apply COW, and we synchronize mapping changes between the generations.

The Linux virtual-memory subsystem manages ASs in two layers: The lower layer is hardware dependent and consists of page directories and tables, which have on AMD64 up to 5 (sparsely-populated) indirection levels. On top of this, *virtual memory allocations (VMAs)* group together the non-connected pages into continuous ranges. VMAs contain information for the page-fault handler (e.g. file backing), swapping, and access control. Together, page directories and the list of VMAs, are kept in the *memory map (MM)*, which is attached to a *thread control block (TCB)*.

While Linux normally has a one-to-one relation between MM and process, we discard this convention and let threads in the same process have different MMs, which are *siblings* of each other. Each AS generation has its own distinct MM, which we keep synchronized with its siblings.

Besides adding a list of all existing siblings to the process, we extended each MM to include a reference to a *master MM*. We use this master MM, which is the process's initial MM and its very first generation, to keep track of all shared memory pages. Furthermore, we use the master MM as a fallback for lazily-instantiated page ranges. Therefore, the master persists until the process exits. It cannot be deleted before, even if no thread currently executes in this generation.

When the user calls `wf_pin()` on a memory region, we mark underlying VMAs as non-shared between generations. We allow pinning only on the granularity of whole VMAs and before the first call to `wf_create()`, when the master MM is the only MM in the process.

On `wf_create()`, we duplicate the calling thread's MM

similar to the `fork()` system call when it creates a new child process: For each VMA of the MM, we copy it and its associated page directories to the newly created sibling MM, while all user-pages are physically shared between generations. While `fork()` marks all user pages as COW, we use COW only for pinned VMAs, while most VMAs behave as shared memory regions, which results in the automatic synchronization of user data between generations. By using Linux's COW mechanism for the pinned regions, we are able to lazily duplicate only those physical pages that are actually modified by the patch. After duplication, we select a new generation ID and insert the MM into the process's sibling list.

When a thread calls `wf_migrate()`, we modify its TCB to point to the respective sibling MM. When the thread returns from the system call, it automatically continues its execution in the selected AS generation. Furthermore, each thread that inhabits a generation increases the reference count of the generation by one. Thereby, we ensure that a generation keeps existing as long as threads execute in this address space, even after the user has instructed us to remove the generation (by calling `wf_delete()`). Only after the last thread leaves a deleted generation, we remove the MM and its page directories.

While the system call interface of WFPATCH is straightforward to implement, its integration with other system calls and the page fault handler requires special attention: As some system calls (e.g., `mmap()`, `mprotect()`, or `munmap()`), change a process's AS, we modified these system calls to apply their effects, as long as they touch shared VMAs, not only to the currently active MM but also to all siblings. However, modifying the protection bits for regions in pinned mappings (via `mprotect()`) affects the current MM only.

We also had to modify the page-fault handler, as Linux allows VMAs and the underlying page directory to become out of sync. For example, within a newly-created anonymous VMA, no pages are mapped in the page directory, but they are lazily allocated and mapped by the page-fault handler. By having multiple sibling MMs, we have to make such lazy page loads visible in all generations, when they happen in a shared VMA. We accomplish this by updating not only the current page directory, but also the page directory of the master MM. Upon page faults, we first search the master MM for lazily loaded pages, before allocating a new page.

In order to avoid race conditions between concurrent system calls that modify a process's AS, we use the master MM as a read-write lock that protects all siblings at once. Normally, the MM linked in the TCB is used for this synchronization, but this is insufficient for WFPATCH to synchronize concurrent accesses. Therefore, we decided to use the master MM as a locking proxy and automatically replaced all MM locks with equivalent lock calls to the master MM by using a Coccinelle [27, 28] script. This replacement alone is responsible for 700 of the 2000 lines of changed source code. For processes that do not have multiple generations, this locking strategy imposes no further overhead as the initial MM is the master MM.

In case a process with mutliple AS generations invokes `fork()`, we clone solely the calling thread's currently active generation and make it the only generation in the AS of the child process. This is sufficient, as `fork()` only copies the currently active thread to the newly created process. In order to maintain COW semantics between the forked AS and all generations of the original AS, we have to mark the appropriate page-table entries of all generations as COW pages (i.e. set the read-only flag) – not only the entries of the two directly involved MMs, as we normally would do. This poses a small overhead when forking processes with multiple generations.

When a COW page gets resolved in an AS with multiple generations, we must ensure that the newly copied page replaces the old shared page in all generations, not just in the current one. Therefore, the page fault handler removes the corresponding page-table entry in all generations and maps the new page into the master MM. The master MM fallback mechanism will fill the siblings' page-table entries again (with the copied page) in case of a page fault.

As the AS generations are technically distinct MMs, the migration of a thread to a new AS generation is treated like a context switch between processes. Each generation gets its own *address-space identifier (ASID)* on the processor. Thus, there is no need for a TLB shootdown on AS migrations. Of course, a TLB shootdown (for all generations) is still necessary if access rights become more restricted.

While our kernel extension is a robust prototype, several features are still missing (e.g., userfaultfd, a mechanism to handle page faults in user space) and some are not extensively tested (e.g., swapping, NUMA memory migration, memory compaction). However, for none of these features, we see any fundamental problem that would conflict with our approach or cause a significant deterioration in the performance of the overall system after adding full support.

3.3 User-Space Library

Our proposed system interface (see Section 3.1) allows a process to create new AS generations, to migrate individual threads, and to delete old generations. In order to utilize this system-call interface for live patching with local quiescence, we built a user-space library around this system-call interface. In the following, we will describe its API as well as its usage in a multi-threaded server with one thread per connection (see Figure 4).

At start, the user initializes and configures our library with `wf_init()`: With `track_threads`, she promises to signal the birth and death of threads such that our library can keep track of all currently active threads and delete old AS generations after the last thread has migrated away. Alternatively, the user can configure a callback that returns the current number

```
int main(void) {
 wf_config_t config = {
  .track_threads = 1,
  .on_migration_start=&f,
 };
 wf_init(config);
 wf_thread_birth();
 signal(RTMIN, sigpatch);
 ...
 while (true) {
   int c = accept();
   spawn_worker(c);
   wf_quiescence();
 }
}
```

```
void worker(int fd) {
  wf_thread_birth();
  while (!done) {
    x = read(fd);
    work(x);
    wf_quiescence();
  }
  wf_thread_death();
}
```

```
void sigpatch(int) {
  char *p;
  p = find_patch();
  wf_load_patch(p);
}
```

Figure 4: Usage of our User-Space Library

of threads. Furthermore, the user can install other callbacks that we invoke at certain points of the migration cycle. In the example, we invoke `f()` when the new AS is ready for migration and, thereby, give the user the possibility to trigger blocked threads in order to speed up the migration phase. With the initialization, the library starts the patcher thread, which pins the text segment, creates new AS generations, and orchestrates the migration phase.

As initiation of live updates and the location of patch files is application specific, we leave this to the user application and only provide a library interface to start the patching application (`wf_load_patch()`). This function instructs the patcher thread to load a binary patch from the file system and apply it in a new AS generation. In our current implementation, `wf_load_patch()` supports ELF-format patches created by Kpatch [30]. These patches are loaded, relocated, and all contained functions are installed in the cloned text segment via unconditional jumps at the original symbol addresses. Furthermore, all references within the patch to unmodified functions, global variables, and shared-library functions are resolved dynamically. Afterwards, the patcher marks the new AS as ready for migration and sleeps until all thread have migrated.

At the thread-local quiescence points, the user has to call `wf_quiescence()` periodically, which checks if a new AS generation is available and ready for migration. If so, the library calls `wf_migrate()` in the context of the current thread and increases the number of migrated threads. After all threads have migrated, the patcher thread is woken, deletes the old AS generation and ends the migration phase.

4 Evaluation

We evaluate WFPATCH with six production-quality infrastructure services on a Linux 5.1 kernel running the Debian 10 Linux distribution (codename "buster", released on 2019-07-10). Table 1 provides a brief overview of the respective Debian packages for OpenLDAP, Apache HTTPD, Memcached, Samba, Node.js, and MariaDB. We use the initial Debian 10 packages and prepare the server executables for dynamic patching with global and local quiescence (Section 4.1). Our goal is to apply all patches published by the Debian maintainers until 2020-05-09 for these binaries with our approach (Section 4.2). This situation mimics a system administrator who maintains a long-running server running one of these services.

For quantitative evaluation, we measure and compare the service latency while applying a binary patch with global and local quiescence (Section 4.3), respectively, as well as the memory and run-time overheads caused by WFPATCH (Section 4.4).

4.1 Implementation of Quiescence

As outlined in Section 2, implementing global quiescence in a complex multi-threaded program can be a difficult undertaking causing three problems in general: Long-running computations (Problem 1) and waiting for I/O (Problem 2) prolong the transition period, which results in deteriorating service quality, while inter-thread dependencies necessitate stopping the threads in an application-specific order to avoid deadlocks (Problem 3). In the following, we describe how we encountered these three problems in our evaluation targets and how they manifest in their structure and fundamental design decisions. Besides the steps we had to take in order to achieve global quiescence, we also describe how we can reach local quiescence for each of the projects we evaluated.

OpenLDAP The OpenLDAP server (`slapd`) uses a listener thread that accepts new connections and dispatches requests as work packages to a thread pool of variable, but limited size ($\leq$ 16 threads). Each work package is processed by a single worker thread, which alternates between computation and blocking I/O until the request is answered.

For global quiescence, we submit a special task to the thread pool. The executing worker pauses all other workers with the built-in pause-pool API, which can only be called from a worker context, and visits a quiescence point on behalf all worker threads. Since the listener thread waits indefinitely for new connections, we need to introduce an artificial timeout (1 second) to provoke quiescence points periodically. For *local* quiescence, we only introduce a quiescence point before the listener waits for a new connection and after a worker thread completes a task.

As worker threads execute client requests as a single task without visiting a quiescence point, complex requests (problem 1), slow client connections (problem 2), and large result sets (problem 2) prolong the barrier-wait time.

Apache The default configuration of the Apache web server (`httpd`) uses the built-in multi-processing module `event`, which implements one dedicated listener thread and a configurable number of worker threads (default: 25). That listener thread handles all new connections, all idle network sockets, and all network sockets whose write buffers are full to

avoid blocking of the worker threads. In its main loop, the listener thread periodically checks for activity on the listening, idle, and full network sockets by using the Linux system call `epoll()` with a timeout of up to 30 seconds, which can cause Problem 2. Once a network socket becomes active, the listener thread unblocks the next free worker thread to serve that socket.

We introduce one quiescence point into each main loop of the listener and worker threads. For global quiescence, however, we have to make sure that the listener thread enters global quiescence after all worker threads have done. Otherwise, some worker threads may block indefinitely because the listener thread cannot unblock them anymore (Problem 3). When returning from global quiescence, the listener's timeout queue needs to be fixed manually to account for the elapsed time spent in global quiescence.

Implementing *local* quiescence in Apache is straightforward by just introducing the same quiescence points without bothering about deadlocks nor timeouts.

Memcached Memcached is event-driven and uses 10 threads in the default configuration: Four worker threads wait for network requests and the completion of asynchronous I/O tasks. One listener thread accepts new connections and wakes up at least every second to update a timestamp variable. Both the workers and the listener use `libevent` to orchestrate event processing. Furthermore, three background threads wait on a condition variable, while two other threads use `sleep()` to wake up periodically with a maximal period of one second.

For global quiescence, we use a built-in notify mechanism to wake up the all workers immediately, even if they are blocking in `libevent`. For the listener thread, we have to use `event_base_loopbreak()` to interrupt the event-processing loop. Unfortunately, this only sets a flag that the listener checks within the aforementioned one-second period. Furthermore, we have to signal the three condition variables to wake up the associated maintenance threads, as they would block indefinitely otherwise. The two sleeping threads will, eventually, reach the quiescence point, but waking them is not necessary to avoid deadlocks. For *local* quiescence, we use the same quiescence points and the same wake-up strategy as for global quiescence.

While the main operation of Memcached is event-driven and, therefore, the threads do not block on I/O operations, the periodic maintenance threads and the listener thread provoke barrier-wait times of up to one second (Problem 2).

Samba For live patching, Samba's `smbd` was especially challenging as it uses a combination of process-based and thread-based parallelization. For each connection, which can live for hours and days if established by a client mount, the process is forked and uses internally a thread pool to parallelize requests. This thread pool shrinks and grows dynamically with the request load, while idling worker threads retire only after a given timeout (1 second). Technically, these workers wait on a condition variable with a one-second timeout and are woken when a listener thread enqueues a received request. In order to issue a patch request, the system administrator has to inform all processes to initiate the patching process.

For global quiescence, we have to signal each worker's condition variable. A woken worker checks whether the barrier is active and visits a quiescence point instead of retiring early as an idle worker. For local quiescence, we just inserted quiescence points after the condition wait and after a received network request.

As each request is limited in size, `smbd` only suffers from problem 2 when workers wait for a send operation to complete. However, as the thread pool dynamically grows to up to 100 threads under heavy load, the overall barrier-wait peaks when the server is most intensely used.

Node.js For asynchronous I/O operations, Node.js spawns one thread that executes a `libuv` loop. For computation, Node.js uses one work queue for immediate tasks executed by a variable number (n) of worker threads, and a second queue for delayed tasks, which is serviced by a dedicated thread. Each worker executes tasks sequentially and offloads I/O to the `libuv` thread.

For binary patching, we introduce quiescence points in the I/O thread and after a worker completes a task. For global quiescence, we submit n empty tasks to the immediate work queue and one task to the delayed work queue. For the `libuv` thread, we had to manually signal a semaphore to prevent deadlocks (problem 3). For *local* quiescence, we only submit one task to the delayed work queue and use the same quiescence points otherwise.

As all computation, including the just-in-time compilation, is dispatched via work queues, a long job (problem 1) will increase the barrier-wait time even though the Javascript execution model is inherently event-driven.

MariaDB MariaDB's `mysqld` supports two thread models: one thread per connection, which is the default, or a pool of worker threads. In both cases, a separate listener thread accepts new connections and passes them to connection or worker threads, and a total of 30 helper threads handle offloaded I/O and housekeeping. We implemented patching support for both thread models.

Judging from its public bug tracker, SQL query evaluation appears to be MariaDB's most error-prone component. We therefore limit the global barrier to threads parsing or executing SQL statements and do not add quiescence points to listener or helper threads. Even so, our global quiescence implementation faces all the three challenges outlined in Section 2.

Slow queries, such as complex SELECT or large INSERT statements, increase the barrier-wait time as threads perform the computation (problem 1) without visiting a quiescence point. Depending on the query and the size of the database, this can lead to excessive wait times.

In both threading variants, idle threads are cached in anticipation of new work before being retired. In one thread

per connection mode, the hard-coded timeout is five minutes; for the thread pool, it defaults to one minute (problem 2). As barrier-wait times of over a minute are unrealistic for any global-quiescence integration, we utilize preexisting functions to wake up all cached threads for patching. We introduce a new global patch variable to distinguish between a wake up due to a new connection, server shutdown, or patching in one thread per connection mode.

MariaDB supports *SQL transactions*, which are an atomic group of SQL statements whose effects are only visible to other connections after the transaction has completed. As MariaDB serializes transactions which access the same data via locks, threads encounter request- and database-induced dependencies (problem 3). If a thread reaches the barrier while holding a transaction lock, other threads that try to get this lock before their next visit at a quiescence point will deadlock. In one thread per connection mode, we handle this by skipping the barrier if the connection holds a transaction lock. For the thread pool, this does not suffice: as each thread handles several connections, waiting on the barrier is forbidden as long as any open transaction is present.

For *local* quiescence, visiting a quiescence point is possible regardless of the transaction state. Apart from that, we use the same quiescence points and wake-up strategies as for global quiescence.

Global vs. Local Quiescence Summarized, we encountered Problem 1 in three projects (OpenLDAP, Node.js, MariaDB), Problem 2 in four projects (OpenLDAP, Memcached, Samba, MariaDB), and Problem 3 in four projects (OpenLDAP, Apache, Node.js, and MariaDB). While Problem 1 and 2 in combination with global quiescence only affect service quality, Problem 3 forced us to introduce different application-specific dead-lock avoidance techniques into our benchmarks. Thereby, we repeatedly experienced set-backs and spurious deadlocks while navigating the often complex web of existing inter-thread dependencies – achieving global quiescence was the hardest part of our evaluation! In contrast, incorporating WFPATCH was straightforward as we only had to identify the local-quiescence points before patch application could start.

4.2 Binary Patch Generation

To demonstrate the applicability of live patching in running user-space programs, we created a set of binary patches for the aforementioned six network services (see Table 1). For each project, we use the current version that is shipped with Debian 10.0 as a baseline against which we apply patches. In Debian, it is common to select one version of a project for a specific Debian release and have the maintainer backport critical patches onto that version.

For five projects (except MariaDB), we systematically inspected the Debian source package for maintainer-prepared patches that touch the source code of the network service. Debian patches reflect critical updates that an expert on the service selected for this specific version. Therefore, we consider these patches as a good candidate set for live patches that a system administrator wants to apply. We also review the subset of patches with a CVE entry to get statistics of highly-critical security updates.

For MariaDB, the source package contains no patches: Debian follows MariaDB releases instead of backporting individual patches. Therefore, we processed all commits in the 10.3 branch of the MariaDB repository, starting with the 10.3.15 release shipped with Debian 10.0. Each set of commits that references a single bug tracker entry classified as *Bug* with a severity of at least *Major* related to `mysqld` is a source patch. As the bug tracker does not reference CVE numbers, we use patches with a severity of at least *Critical* instead.

From these source-code patches, we manually select those which only influence the .text segment and do not alter data structures or global variables, as such patches are currently out of scope for our mechanism. In Table 1, we see that most patches that are hand-selected by a maintainer are text-only patches; for CVE patches, the correlation is even higher. For MariaDB, where we have a large set of critical patches, 91 percent of the patches exclusively modify the program logic. We therefore conclude that a mechanism which supports live patching with a restriction to code-only changes is nevertheless a useful contribution for keeping running services up to date.

As patch generation, in contrast to patch application, is not among our intended contributions, we use the Kpatch toolchain, which was developed for live-updating the Linux kernel, to prepare binary patches from source code changes. Unfortunately, due to shortcomings in Kpatch, we could not create binary patches for all text-only changes. Especially MariaDB and Node.js, which are implemented in C++, show a low success rate. In the lower half of Table 1 we summarize, over all generated binary patches, the average number of changed object files, modified function bodies, and the size of each patch text segment.

We verified our mechanism by applying each patch into the corresponding service while processing requests. We successfully applied all binary patches generated by Kpatch with our user-space library using thread migration at local quiescence points.

In total, we successfully applied 33 different binary patches including 15 CVE-relevant patches. For OpenLDAP, Apache, and Samba, we were able to apply all generated patches sequentially into the running process. This was not possible for MariaDB because the patches are not applicable to a common base version due to the amount of patches that we could not generate with Kpatch. Making the patches applicable sequentially in MariaDB would have meant to backport them to the initial version, like the Dabian maintainers did for the other projects.

		OpenLDAP (slapd)	Apache (httpd)	Memcached	Samba (smbd)	MariaDB* (mysqld)	Node.js
Release		2.4.47	2.4.38	1.5.6	4.9.5	10.3.15	10.19.0
All Patches (CVE)	[#]	13 (2)	10 (10)	1 (1)	2 (2)	74 (26)	4 (0)
.text Only (CVE)	[#]	9 (2)	7 (7)	1 (1)	2 (2)	67 (24)	4 (0)
kpatch'able (CVE)	[#]	9 (2)	7 (7)	1 (1)	2 (2)	16 (5)	0 (0)
⌀Mod. Files	[#]	1.11	1.71	1	1	1.19	–
⌀Mod. Functions	[#]	3.67	13.71	1	5.5	2.94	–
⌀Patch Size	[KiB]	13.02	56.94	43.91	9.23	15	–

* For MariaDB, no Debian patches were available and MariaDB maintainers do not relate bugs to CVEs. We instead took patches with severity $\geq$ *Major* from the project's bug tracker as base; numbers in brackets denote patches with severity $\geq$ *Critical*.

Table 1: Evaluation projects and patches (of which CVE-related) since Debian 10.0 release

4.3 Request Latencies

In order to quantify the service quality benefits of local quiescence and incremental thread migration over the barrier method, we perform an end-to-end test for our selected projects. For each project, we define a benchmark scenario and measure the end-to-end request latencies encountered on the client side, while we (a) generate new AS generations and migrate threads, or (b) stop all threads at a global barrier. For this, we extended our user-space library to also support global-quiescence states via the barrier method. We periodically send patch requests to the same process and skip the actual text-segment modification in these tests, while still inducing barrier-wait times on the one side and AS-creation overheads on the other side. Thereby, we achieve a high coverage of different program states at patch-request time, while keeping the comparison fair.

All experiments are conducted on a two machine setup. The server process runs on a 48-core (96 hardware threads) Intel Xeon Gold 6252 machine clocked at 2.10 GHz with 374 GiB of main memory. The clients execute on a 4-core Intel Core i5-6400 machine running at 2.70 GHz with 32 GiB of main memory. Both machines are connected by a Gigabit link in a local-area network.

On the server side, we start the service, wait 3 seconds for the clients to come up and then trigger a local-quiescence migration or global-quiescence barrier sync every 1.5 seconds. By this patch-request spreading, the impact of the barrier method can cool down before the next cycle starts. On the client side, we measure the end-to-end latency of each request. In total, we simulate at least 1000 patch requests for each benchmark.

For OpenLDAP, 200 parallel client connections send LDAP searches that result in 50 user profiles from a database with 1000 records. For Apache, we use ApacheBench to download a 4 MiB sample file 50,000 times using 10 parallel connections; due to the shared Gigabit link, a download takes about 350 ms when no threads are blocked on the global quiescence barrier. For Memcached, 50 client connections request a random key from a pool of 1000 cached objects of 64 KiB. For MariaDB, which we operate in the one-thread-per-connection mode, four sysbench `oltp_read_only` connections continuously perform transactions with five simple SELECT statements, while four background connections – whose latency we do not monitor – execute transactions with 2000 statements. For Node.js, we developed an example web service that encodes a request parameter in a QR-code, wraps it in a PDF, and sends the resulting "ticket" back to the client. We use the `wrk` tool to simulate 10 parallel clients that repeatedly request a new ticket. For Samba, we mount the exported file system on the client machine (`mount.cifs`) and use the sysbench `fileio` benchmark with 32 threads, a block size of 16 KiB, and an R/W ratio of 1.5 to measure file I/O latencies.

Please be aware that these scenarios are chosen as examples to demonstrate the possible impact of barrier synchronization. Resulting latencies are highly dependent on the workload and can be smaller, but also vastly larger in other scenarios. For example, by executing long-running SQL queries on MariaDB or downloading large files from an Apache server, the barrier-wait times, and therefore the latency of the global-quiescence method, can be increased arbitrarily.

Figure 5 shows latency histograms (with logarithmic y axis) for local and global quiescence, as well as the 99.5 response-time percentile. In all benchmarks, we see a significant increase in tail latency which ranges from a factor of 0.97× (Node.js) to 41× for MariaDB. While the results for OpenLDAP, MariaDB, and Samba directly show the latency impact of a global barrier, the other results require explanations. For Memcached, three out of ten threads perform one-second waits, resulting in latencies of up to one second. For Apache, local quiescence shows a narrow latency distribution with the predicted peak at 350 ms while global quiescence shows a broadened distribution. This is due to the benchmark's network-bound nature: the last worker to reach the barrier enjoys the unshared 1 Gigabit link to finish its last request, while all requests arriving after the patch request are impacted by the barrier-wait time. In Node.js, the percentiles

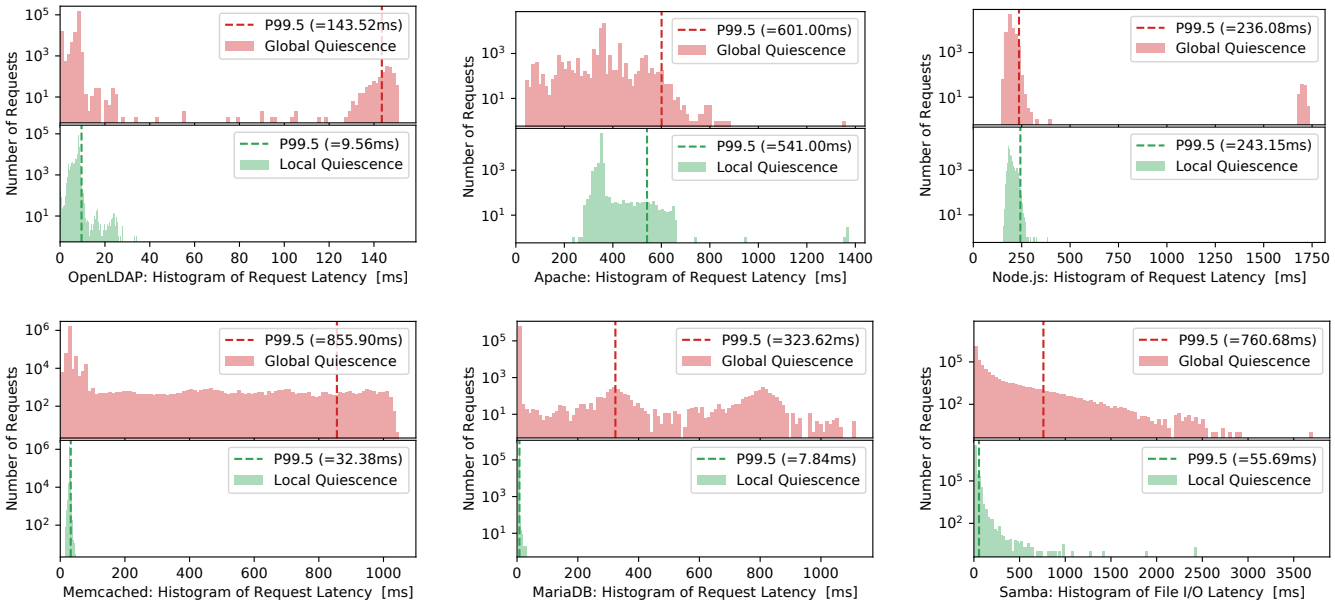


Figure 5: Request Latencies during Live Patching

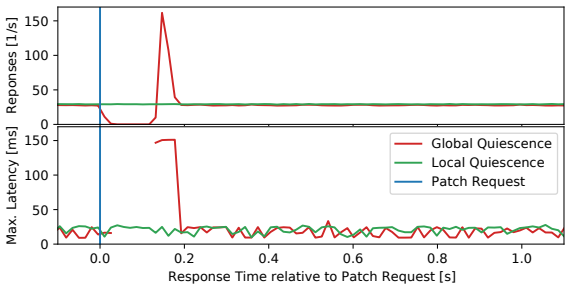


Figure 6: OpenLDAP Response Rates during Quiescence

are almost equal as the longest encountered barrier-wait time (18 ms) is still shorter than the average request duration's jitter (193±53 ms). However, we observe individual barrier-wait times of more than 1.5 seconds.

For a deeper understanding of the encountered service quality directly after a patch request, we analyze OpenLDAP responses during 1000 patch requests. We correlate each received response to the previous patch request and plot them according to their relative receive time; zero being the patch request. Figure 6 shows response rate and maximum observed latency. After a patch request, the response rate in the global-quiescence case rapidly decreases, while the latency stays at its normal value. After the workers reach the barrier, no responses are recorded until the listener has reached the barrier. After global quiescence is reached, `slapd` ramps up again and processes the request backlog built up in the meantime. This causes the response rate to spike, but those responses are so late that we see a significant latency increase before the service returns to normal operation. With WFPATCH, no impact, neither on the response rate nor on the maximum latency, can be observed.

4.4 Memory and Run-Time Overheads

For each patch application, our kernel extension duplicates the MMU configuration, creates a new AS generation, and performs one AS switch per thread in order to migrate it to the new generation. To quantify the impact of these operations, we measure the MMU configuration size and perform run-time micro benchmarks of AS creation and switching times for each server application. We run the benchmarks under load (see Section 4.3) to provoke disturbance and lock contention in the kernel.

We measure the memory overhead caused by duplicate MMU configurations by sequentially applying as many patches as possible. In Table 2, we report the difference in MMU configuration size before and after the patch application. As the other data-structure additions required for our extension are negligible in size, this is the total memory overhead during patch application. Due to the non-deletable master MM (see Section 3.2), this overhead becomes permanent for patched processes: starting with the first additional generation, we carry the load of this additional MM. We do not introduce a memory overhead for processes which do not use AS generations.

For the run-time overhead, we perform two micro benchmarks. (1) The patcher thread creates a new AS generation and immediately destroys it. (2) The patcher thread migrates back and forth between two AS generations (2 switches). We

	Memory [KiB]	Runtime Penalty Create [µs]	Switch [µs]
OpenLDAP	412	298±47	7±7
Apache	680	429±17	7±6
Memcached	132	88±23	7±6
MariaDB	516	1339±38	7±6
Node.js	1808	2171±139	8±7
Samba	256	672±54	5±5

Table 2: Address Space Management Overhead

	Upstream [µs]	WFPATCH [µs]
(a) Anonymous Mapping	0.40±0.12	0.42±0.15
(b) File Mapping	0.50±0.14	0.50±0.15
(c) Read Fault	0.87±0.18	0.87±0.20
(d) Write Fault	1.23±0.29	1.25±0.32
(e) COW Fault	1.79±0.35	1.81±0.39

Table 3: Steady-State Run-Time Overhead

execute each scenario a million times in a tight loop and report, in Table 2, the average operation time alongside its standard deviation. We see that the creation and destruction of AS generations scales with the size of the process's virtual address space. Only for Samba and MariaDB, the creation overhead is, compared to the MM size, disproportionately higher than for the other four benchmarks. This is caused by a higher number of file-backed VMAs in Samba and MariaDB that take longer to duplicate. The `wf_migrate()` call is a constant-time operation.

In the implementation of our approach, we tried to minimize overhead for applications that do not use WFPATCH. Memory consumption overhead is limited to few additional fields in the thread control block (2 pointers + 2 integer fields), the memory map (3 pointers), and the structure that represents a memory mapping (1 boolean field). In terms of run-time overhead, WFPATCH adds code in two critical places in the kernel: the mapping modification functions and the page-fault handler. In order to assess the run-time impacts, we performed micro benchmarks on our modified kernel and on an upstream kernel with the same version and configuration. To evaluate mapping modifications, we map and unmap either (a) an anonymous memory region or (b) a file mapping and measure the time of the `mmap()` system call. The results do not show a significant difference between the kernels (see Table 3). For page faults, we issue (c) a read operation or (d) a write operation on a previously untouched portion of an anonymous mapping. To also capture (e) copy-on-write resolution, we write to a page that is also mapped by a forked process. Each of the five measurements was repeated 10 million times.

5 Discussion

Benefits of Local Quiescence The main benefit of patching threads individually is the simplified establishment of quiescence and the avoidance of a global barrier that causes a deterioration in performance. Thereby, WFPATCH provides latency hiding for Problem 1 and 2 (Section 4.1) and mitigates Problem 3.

Nevertheless, in the light of the rare event of applying a live patch to an application, the overhead and tail latency of global quiescence may seem negligible. However, the benchmarks presented in Section 4.3 do not necessarily represent a real-world or worst-case scenario: We use a single client machine with a fast, stable, and reliable local network connection to the server. Furthermore, we aimed for a controlled and uniformly distributed load pattern for the sake of reproducibility and in order to fairly compare the relative impact of global vs. local quiescence. In a real-world scenario, connection latencies will vary wildly or may be even under control by an active attacker. As barrier-synchronized global quiescence couples the progress of all threads in the system, it is much more prone to such latency variations – the latency impact is dictated by the slowest (in case of an attacker: stalling) thread to reach the barrier. With WFPATCH and local quiescence, all other threads will not only continue working, but also have the patch applied immediately. Even if a thread stalls forever, the only damage is an AS generation that will never get freed, while the patched server continues to answer requests.

Lightweight AS Generation For the lightweight AS generation, our current implementation copies the whole MM in `wf_create()`, including VMAs and the page directories. This leads to the differing memory and creation overheads that we observed for our benchmark scenarios (Table 2).

While we consider these overheads as reasonable for the purpose, they could nevertheless be reduced further if we implement the different generations to share parts of their page-directory structure. This is possible for shared VMAs, as the underlying page tables always reference the same physical pages. In fact, we currently even pay for not sharing them by extra efforts to keep page tables synchronized among AS generations via the master MM. However, VMAs cover page ranges with arbitrary start/end index, while the page-directory tree covers page rages on a power-of-two basis, so implementing such sharing is not trivial. To the best of our knowledge, Linux itself does not employ page-table sharing between address spaces, even though this would probably be beneficial for the implementation of the fork system call.

Code Complexity The current implementation of WFPATCH adds a certain amount of complexity to the kernel (see Section 3.2). This stems from its interaction with the already-complex kernel memory-management subsystem. One reason is that Linux targets numerous different architectures and exploits most of their individual capabilities. Secondly, the

kernel itself provides many features and often chooses performance over simplicity (e.g., fine granular page table locking or code duplication in the mapping functions). Apart from that, WFPATCH's complexity is also caused by the tight connection between address spaces and processes in Linux. As the idea of AS generations itself is straightforward, the complexity of our kernel extension could be reduced significantly if we decoupled the two concepts of address spaces and processes in general. That would not only serve our approach, but may even promote other ideas and development [21, 9], such as the decoupling between threads and processes did.

Applicability The general applicability of WFPATCH is potentially limited by (a) the restriction to .text/.rodata-patches only and (b) the preparation of the respective target program. With respect to (a) this depends on the intended use case: We consider WFPATCH currently as an approach to apply hot fixes to a server process under heavy load – in order to prolong the time it needs to be restarted until the next maintenance window. For this use case, our results show that the vast majority of patches (87%) are .text-only and, therefore, applicable; for critical patches (CVE mitigations) this number is even higher (88%). Regarding (b), the WFPATCH user-space library simplifies the preparation of the target program to support hot patching, but like in other approaches that support multi-threaded applications, it is up to the developer to identify and model the respective safe points to apply a patch. With WFPATCH, however, it becomes significantly easier to find these points as they need to be only locally quiescent. In our evaluation, the hardest part of integrating WFPATCH into the six multi-threaded server programs was the global barrier we needed solely for the comparison between local and global quiescence.

Soundness and Completeness Proving the soundness of a dynamic update is an undecidable problem [14], even though type checking and static analysis can help to mitigate the situation in some cases [1]. With WFPATCH, we have the additional complexity of *incomplete patches*, that is, some threads still execute the old code, while others already use the patched version. This, however, imposes additional correctness issues only if the code change actually influences inter-thread data/control dependencies, such as the implementation of a producer–consumer protocol. In practice, this is a rare situation – none of the analyzed 90 .text-only patches fell into this category. Nevertheless, a possible solution in such cases would be to gradually give up the wait-free property by implementing *group quiescence* among the dependent threads, while all other threads can still migrate wait-free at their local quiescence point. Compared to global quiescence, group quiescence would still be less debilitating for overall response time and easier to implement in a deadlock-free manner.

In general, if some thread has not yet passed its point of local quiescence, it is either blocking somewhere in an I/O or still actively processing a request that arrived before the patch was triggered. In both cases, it is at most this one request that may still be processed using the old version. This would also be the case with global quiescence – only that with global quiescence based on barriers all other threads have to wait (see Figure 6); if global quiescence is determined by probing for a safe state (such as in Ksplice [3]), the other threads continue processing requests using the unpatched version. If the respective thread hangs forever, global quiescence based on barriers would result in a deadlock, while with probing the patch would never get applied. With WFPATCH, the patch will be applied as far as possible: All new requests will be processed with the new code – a server may even be patched while under an active DDOS attack. Technically, an incomplete patch means that the process will stay in two (or even more) ASs forever.

Overall, local and global quiescence make a different tradeoff between correctness requirements and ease of patch applicability: While applying patches with global quiescence requires less upfront thought about the correctness of a patch as it provokes no transition period, it may be hard or even impossible to introduce the patch in the system. On the other hand, although it is harder to show that a patch is suitable for local-quiescence patching, finding local-quiescence points is easier and patch application has only minimal impact on the system's operation. We believe that many time-critical updates (e.g., additional security checks) have such a localized impact on the code that the guarantees of local-quiescence patching are sufficient for a large number of changes.

Generalizability For the sake of simplicity, we chose to adapt the Kpatch binary-patch creation for our evaluation and implemented a loader for such patches for user-space programs (Section 3.3). Thereby, we also inherit the limitations of Kpatch regarding granularity and installation of patches: Patches work at the granularity of functions; they are installed by placing a jump at the original symbol address to redirect the control flow to the patched version. This bears some overhead, but is arguably the most widespread technique to apply run-time patches [1, 23, 3, 29, 30, 5, 6]. Furthermore, only quiescent (inactive) functions can be patched. While this limitation is a lot less problematic with WFPATCH due to the fact that quiescence is reduced to local quiescence (inactive in the currently examined thread), it nevertheless prevents patching of top-level functions.

It is important to note, though, that these are restrictions of the employed patching mechanism, not of its wait-free application offered by WFPATCH, which is the main contribution of this work. Integration with more sophisticated patching methods [17, 15, 22] could mitigate these limitations while keeping the WFPATCH benefits. For instance, UpStare can patch active functions by an advanced *stack reconstruction* technique [22]. Hence, it does not require quiescence, but nevertheless has to halt the whole process for patch application and reconstruction of all stacks. In conjunction with WFPATCH, this expensive undertaking could be performed in the background while other threads continue to make progress.

Data Patching While our toolchain already supports the introduction of new data structures and global variables, we currently do not support patches that change existing data-structures or the interpretation of data objects. Such patches are generally difficult [36] as a transform function that migrates the system state to the new representation must be applied to all modified objects in existence. Current live-patching systems rely on the developer to supply these transform functions [17, 15], while language-oriented methods for semi-automated transformer generation exists [20, 18, 25].

With local quiescence, state transfer becomes more difficult as two threads that touch the same data can execute in different patching states. Therefore, an extension to data patches would require bidirectional transform functions that are able to migrate program state back and forth as needed. MMU-based object migration on read and write accesses via page faults can be used to trigger the migration of individual objects between AS generations. Similar mechanisms are used to provide virtual shared memory on message-passing architectures [2]. However, for thread-local state only a unidirectional transform function is required.

Other Applications In a nutshell, WFPATCH provides means for run-time binary modifications in the background, which can then be applied wait-free to individual threads. Besides run-time binary patching, the fundamental mechanism could be useful for many further usage scenarios.

For example, every just-in-time (JIT) compiler has to integrate newer, more optimized versions of functions into the call hierarchy while the program is executing. With WFPATCH, the JIT could prepare complex changes and rearrangements across multiple functions in the background in a new AS generation and then apply them, without stopping user threads, by migrating the benefiting threads incrementally to the updated AS. Furthermore, as our kernel extension supports an arbitrary number of AS generations, the JIT could provide specialized thread-local function variants with the same start address, keeping all function pointers valid.

In a similar manner, an OS kernel could transparently apply path-specific kernel modifications [31] on a per-thread basis. For example, the kernel could use a different IRQ subsystem that is only used if a thread with real-time priority gets interrupted.

AS generations can not only be used to provide a differing code views between threads, but also data views. This can be employed to provide isolation for security and safety purposes. For example, a server application could make encryption keys only be present in a special AS generation; the other generations would have an empty mapping in this place. Even individual threads could live in their own AS generations in order to keep sensible data private but share all the other mappings with their sibling threads. The major benefit compared to using `fork()` with distinct processes is that all mappings are shared by default and modifications to the mapping are implicitly synchronized – the address spaces do not diverge. Moreover, threads can easily switch back and forth between generations. Litton et al. [21] made a similar suggestion in form of thread-level address spaces, which, however are not synchronized, thus being similar to `fork()` in this respect.

In general, WFPATCH is able to provide classical cross-cutting concerns (debugging, tracing, logging) with a thread-local view of the text segment. For example, a debugger may limit the effect of trace- and breakpoints to the actually debugged threads or use the unoptimized program only during the debugging session. Also, the user could enable tracing, logging, assertions, or behavioral sanitizers (e.g., Clang's UBSan) for individual threads.

6 Related Work

Dynamic patching of OS kernels has a long history in research [13, 4, 5, 12] and is now actually used in production systems [3, 29, 30]. In contrast, the suggested frameworks to patch user-level processes [20, 25, 6, 22, 17, 15, 12] are still not broadly employed.

The DAS [13] operating system incorporated an early run-time updating solution on module-level granularity. It requires absolute quiescence of a module to be patched, realized by locks. K42 [4] exploits its strict object-oriented design to enable live kernel updates. The event-driven nature with short-lived and non-blocking threads makes it relatively easy to define a safe state for concurrent patching.

The Proteos [12] microkernel provides built-in means for process-level live updates based on automatic state transfer. Like our wait-free patching technique, they employ MMU-based address spaces, but unlike our approach the goal is not a seamless thread-by-thread migration. Instead, the process is halted during the update procedure, while the separate address space provides for an easy rollback.

Most live-patching frameworks work on function-level granularity [1, 23, 3, 29, 30, 5, 6], which can be considered as a natural scope for changes while still providing for relatively fine-grained updates. A patched version of the function is loaded and installed via placing a trampoline jump at the beginning of the old function body (function indirection). Barrier blocking is the classical way to reach global quiescence to safely apply the trampoline. Ksplice [3] avoids this by polling for global quiescence instead: The whole kernel is repeatedly stopped and checked for a safe state before the function indirection gets installed. While this avoids a global barrier, all threads have nevertheless to be halted for the check and to apply the patch. Furthermore, probing is an unbounded operation, so the patch may be applied late or never.

DynAMOS [23] and kGraft [29] also avoid global barriers by extending the function indirection method: By (atomically) placing additional redirection handlers between the trampoline and the jump target, they can decide on a per-call basis which version of a function (original/updated) should be used. This has some similarities to our address-space migration

technique as in both methods the patched and the unpatched universe coexist while the transition is in progress; however, in contrast to our approach, the redirection method induces a performance penalty in this phase. Atomicity is reached by rerouting the call through debug breakpoints during the patch process; on SMP systems this furthermore requires IPIs to all other cores to flush instruction caches. This approach is limited to patching on function granularity and has only been explored for kernel-level patching, whereas WFPATCH targets user-level processes and allows for arbitrary large (or small) in-place binary modifications, which in principle also includes changes to (read-only) data.

LUCOS [5] tries to solve this by requiring the to-be-patched kernel to run inside a modified XEN hypervisor, which is able to atomically install trampoline calls by halting the VM. The virtualization layer is also used to enable page-granularity state synchronization between the different versions of a function. POLUS [6] brings this idea to user space and relies on the underlying operating system (ptrace, signals and mprotect) instead of a hypervisor. Again, all threads are halted while the trampoline gets installed.

Ginseng [25] makes use of source-to-source compilation in order to prepare C programs for dynamic updating. It inserts indirection jumps for every function call and every data access, but does not support multi-threaded programs. Function indirections are also used by many other language-oriented dynamic-variability methods, such as dynamic aspect weaving [7, 10, 34] or function multiverses [33], which, however, do not address quiescence in multi-threaded environments.

Ekiden [17] and Kitsune [15] provide dynamic updates by replacing the whole executable code and transferring all program state at dedicated *update points*, which constitute points of global quiescence implemented by barriers in the case of multi-threading. UpStare [22] goes one step further by allowing run-time updates at arbitrary program states, enabled by its *stack reconstruction* technique. However, updating multi-threaded programs is also based on halting all threads. The authors even suggest inserting the respective checks in long-lived loops and to avoid blocking I/O.

Duan et al. present a comprehensive solution for patching vulnerable mobile applications on the binary level [8]. However, patching takes place when the program starts and not during later run time.

The idea of decoupling address spaces and processes has also been described before: El Hajj et al. [9] provide freely switchable address spaces in order to enlarge virtual memory and to support persistent long-lived pointers. However, they do not target live patching and their address spaces are intended to be decoupled from each other, whereas WFPATCH provides extra means to synchronize most regions among address space generations.

Litton et al. [21] allow for multiple "light-weight execution contexts" (lwC) per process and the possibility for threads to switch between them. After creation, where the file-descriptor table and the AS are copied (like fork), lwCs are decoupled entities and can diverge significantly from each other. In contrast, our AS generations offer a gradually differing view of the same AS without decoupling other parts of the execution context (i.e. file-descriptor tables). Thereby, all threads retain a synchronized view of process state, which is necessary for incremental thread migration.

7 Conclusion

WFPATCH provides a wait-free approach to apply live code patches to multi-threaded processes without "stopping the world." The fundamental principle of WFPATCH is that a code change is not applied to the whole process at once, which requires a state of global quiescence to be reached by all threads simultaneously, but incrementally to each thread individually at a thread-specific state of local quiescence. Hence, (1) no thread is ever halted, (2) a single hanging thread cannot delay or even prevent patching of all other threads, and (3) the implementation gets easier as quiescence becomes a (composable) local property. The incremental migration is provided by means of multiple generations of the virtual address space within the updated process. After preparation of an updated address space, threads switch generations at their local quiescence points, while they are still able to communicate with threads in other generations via shared memory mappings.

We implemented WFPATCH as a Linux 5.1 kernel extension and a user-space library, and evaluated our approach with six major network services, including MariaDB, Apache and Memcached. While live patching at points of global quiescence with a barrier increases the tail-latency of client requests by up to a factor of 41×, we could not observe any disruption in service quality when live patches were applied wait-free with WFPATCH. In total, we successfully applied 33 different binary patches into running programs while they were actively servicing requests; 15 patches had a CVE number or were other critical updates.

WFPATCH brings us closer to an ideal live patching solution for multi-threaded applications by solving the response-time issue with a latency hiding patch-application mechanism. This opens further research opportunities on advanced patching techniques.

Acknowledgments

We thank our anonymous reviewers and our shepherd Andrew Baumann for their constructive feedback and the efforts they made to improve this paper. We also thank Lennart Glauer for his work on an early WFPATCH prototype.

This work was supported by the German Research Council (DFG) under the grants LO 1719/3, LO 1719/4, SP 968/9-2.

The source code of WFPATCH *and the evaluation artifacts are available at:*
https://www.sra.uni-hannover.de/p/wfpatch

References

[1] ALTEKAR, G., BAGRAK, I., BURSTEIN, P., AND SCHULTZ, A. Opus: Online patches and updates for security. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), SSYM '05, USENIX Association, pp. 19–19.

[2] APPEL, A. W., AND LI, K. Virtual memory primitives for user programs. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* (1991), pp. 96–107.

[3] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: automatic rebootless kernel updates. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2009 (EuroSys '09)* (New York, NY, USA, Mar. 2009), J. Wilkes, R. Isaacs, and W. Schröder-Preikschat, Eds., ACM Press, pp. 187–198.

[4] BAUMANN, A., HEISER, G., APPAVOO, J., SILVA, D. D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005), pp. 279–291.

[5] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P.-C. Live updating operating systems using virtualization. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), VEE '06, ACM, pp. 35–44.

[6] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P.-C. Polus: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 271–281.

[7] DOUENCE, R., FRITZ, T., LORIANT, N., MENAUD, J. M., DEVILLECHAISE, M. S., AND SUEDHOLT, M. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)* (Chicago, Illinois, Mar. 2005), P. Tarr, Ed., ACM Press, pp. 27–38.

[8] DUAN, R., BIJLANI, A., JI, Y., ALRAWI, O., XIONG, Y., IKE, M., SALTAFORMAGGIO, B., AND LEE, W. Automating patching of vulnerable open-source software versions in application binaries. In *2019 Network and Distributed System Security Symposium (NDSS 2019)* (2019).

[9] EL HAJJ, I., MERRITT, A., ZELLWEGER, G., MILOJICIC, D., ACHERMANN, R., FARABOSCHI, P., HWU, W.-M., ROSCOE, T., AND SCHWAN, K. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, Association for Computing Machinery, p. 353–368.

[10] ENGEL, M., AND FREISLEBEN, B. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)* (Chicago, Illinois, Mar. 2005), P. Tarr, Ed., ACM Press, pp. 51–62.

[11] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug. 2004), 5–.

[12] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and automatic live update for operating systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)* (New York, NY, USA, 2013), ACM Press, pp. 279–292.

[13] GOULLON, H., ISLE, R., AND LÖHR, K.-P. Dynamic restructuring in an experimental operating system. *IEEE Transactions on Software Engineering SE-4*, 4 (1978), 298–307.

[14] GUPTA, D., JALOTE, P., AND BARUA, G. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering 22*, 2 (1996), 120–131.

[15] HAYDEN, C. M., SAUR, K., SMITH, E. K., HICKS, M., AND FOSTER, J. S. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM Trans. Program. Lang. Syst. 36*, 4 (Oct. 2014), 13:1–13:38.

[16] HAYDEN, C. M., SMITH, E. K., HARDISTY, E. A., HICKS, M., AND FOSTER, J. S. Evaluating dynamic software update safety using systematic testing. *IEEE Transactions on Software Engineering 38*, 6 (2012), 1340–1354.

[17] HAYDEN, C. M., SMITH, E. K., HICKS, M., AND FOSTER, J. S. State transfer for clear and efficient runtime updates. In *2011 IEEE 27th International Conference on Data Engineering Workshops* (Apr. 2011), pp. 179–184.

[18] HICKS, M., MOORE, J. T., AND NETTLES, S. Dynamic software updating. *SIGPLAN Not. 36*, 5 (May 2001), 13–23.

[19] HSU, T. C.-H., BRÜGNER, H., ROY, I., KEETON, K., AND EUGSTER, P. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the 12th*

European Conference on Computer Systems (EuroSys '17) (2017), ACM, pp. 468–482.

[20] LEE, I. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin-Madison, 1983.

[21] LITTON, J., VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., GARG, D., BHATTACHARJEE, B., AND DRUSCHEL, P. Light-weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 49–64.

[22] MAKRIS, K., AND BAZZI, R. A. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX '09, USENIX Association, pp. 31–31.

[23] MAKRIS, K., AND RYU, K. D. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)* (New York, NY, USA, Mar. 2007), T. Gross and P. Ferreira, Eds., ACM Press, pp. 327–340.

[24] MEENA, J. S., SZE, S. M., CHAND, U., AND TSENG, T.-Y. Overview of emerging nonvolatile memory technologies. *Nanoscale research letters 9*, 1 (2014), 526.

[25] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 72–83.

[26] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[27] PADIOLEAU, Y., LAWALL, J. L., MULLER, G., AND HANSEN, R. R. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)* (New York, NY, USA, Mar. 2008), ACM Press.

[28] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J. L., AND MULLER, G. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)* (New York, NY, USA, 2011), ACM Press, pp. 305–318.

[29] PAVLÍK, V. kgraft: Live patching of the linux kernel, 2014. https://www.suse.com/media/presentation/kGraft.pdf, visited 2019-08-05.

[30] POIMBOEUF, J., AND JENNINGS, S. Introducing kpatch: Dynamic kernel patching, 2014. https://rhelblog.redhat.com/2014/02/26/kpatch, visited 2019-08-05.

[31] PU, C., MASSALIN, H., AND IOANNIDIS, J. The Synthesis kernel. *Computing Systems 1*, 1 (1988), 11–32.

[32] REDISLAB. Redis, 2019. http://redis.io, visited 2019-07-21.

[33] ROMMEL, F., DIETRICH, C., RODIN, M., AND LOHMANN, D. Multiverse: Compiler-assisted management of dynamic variability in low-level system software. In *Fourteenth EuroSys Conference 2019 (EuroSys '19)* (New York, NY, USA, 2019), ACM Press.

[34] SCHRÖDER-PREIKSCHAT, W., LOHMANN, D., GILANI, W., SCHELER, F., AND SPINCZYK, O. Static and dynamic weaving in system software with AspectC++. In *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS '06) - Track 9* (2006), Y. Coady, J. Gray, and R. Klefstad, Eds., IEEE Computer Society Press.

[35] SELTZER, M., MARATHE, V., AND BYAN, S. An NVM carol: Visions of nvm past, present, and future. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 15–23.

[36] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, I. Mutatis mutandis: Safe and predictable dynamic software updating. In *ACM SIGPLAN Notices* (01 2005), vol. 40, pp. 183–194.



Testing Database Engines via Pivoted Query Synthesis

Manuel Rigger Zhendong Su
Department of Computer Science, ETH Zurich

Abstract

Database Management Systems (DBMSs) are used widely, and have been extensively tested by fuzzers, which are successful in finding crash bugs. However, approaches to finding *logic bugs*, such as when a DBMS computes an incorrect result set, have remained mostly untackled. To this end, we devised a novel and general approach that we have termed *Pivoted Query Synthesis*. The core idea of this approach is to automatically generate queries for which we ensure that they fetch a specific, randomly selected row, called the *pivot row*. If the DBMS fails to fetch the pivot row, the likely cause is a bug in the DBMS. We tested our approach on three widely-used and mature DBMSs, namely SQLite, MySQL, and PostgreSQL. In total, we found 121 unique bugs in these DBMSs, 96 of which have been fixed or verified, demonstrating that the approach is highly effective and general. We expect that the wide applicability and simplicity of our approach will enable improving the robustness of many DBMSs.

1 Introduction

Database management systems (DBMSs) based on the relational model [10] are a central component in many applications, since they allow efficiently storing and retrieving data. They have been extensively tested by random query generators such as SQLsmith [45], which have been effective in finding queries that cause the DBMS process to crash (*e.g.*, by causing a buffer overflow). Also fuzzers such as AFL [2] are routinely applied to DBMSs. However, these approaches cannot detect *logic bugs*, which we define as bugs that cause a query to return an incorrect result, for example, by erroneously omitting a row, without crashing the DBMS.

Logic bugs in DBMSs are difficult to detect automatically. A key challenge for automatic testing is to come up with an effective *test oracle*, that can detect whether a system behaves correctly for a given input [21]. In 1998, Slutz proposed to use *differential testing* [33] to detect logic bugs in DBMSs, by constructing a test oracle that compares the results of a query on multiple DBMSs, which the author implemented in a tool RAGS [46]. While RAGS detected many bugs, differential testing comes with the significant limitation that the systems under test need to implement the same semantics for a given input. All DBMSs support a common and standardized language *Structured Query Language (SQL)* to create, access, and modify data [8]. In practice, however, each DBMS provides a plethora of extensions to this standard and deviates from it in other parts (*e.g.*, in how `NULL` values are handled [46]). This vastly limits differential testing, and also the author stated that the small common core and the differences between different DBMSs were a challenge [46]. Furthermore, even when all DBMSs fetch the same rows, it cannot be ensured that they work correctly, because they might be affected by the same underlying bug.

To efficiently detect logic bugs in DBMSs, we propose a general and principled approach that we termed *Pivoted Query Synthesis (PQS)*, which we implemented in a tool called SQLancer. The core idea is to solve the oracle problem for a single, randomly-selected row, called the *pivot row*, by synthesizing a query whose result set must contain the pivot row. We synthesize the query by randomly generating expressions for `WHERE` and `JOIN` clauses, evaluating the expressions based on the pivot row, and modifying each expression to yield `TRUE`. If the query, when processed by the DBMS, fails to fetch the pivot row, a bug in the DBMS has been detected. We refer to this oracle as the *containment* oracle.

Listing 1 illustrates our approach on a test case that triggered a bug that we found using the containment oracle in the widely-used DBMS SQLite. The `CREATE TABLE` statement creates a new table `t0` with a column `c0`. Subsequently, an index is created and three rows with the values `0`, `1`, and `NULL` are inserted. We select the pivot row `c0=NULL` and construct the random `WHERE` clause `c0 IS NOT 1`. Since `NULL IS NOT 1` evaluates to `TRUE`, we can directly pass the query to the DBMS, expecting the row with value `NULL` to be contained in the result. However, due to a logic bug in the DBMS, the partial index was used based on the incorrect assumption that `c0 IS NOT 1` implied `c0 NOT NULL`, resulting in the pivot row

Listing 1: Illustrative example, based on a *critical* SQLite bug. The check symbol denotes the expected, correct result, while the bug symbol denotes the actual, incorrect one.

```
CREATE TABLE t0(c0);
CREATE INDEX i0 ON t0(1) WHERE c0 NOT NULL;
INSERT INTO t0(c0) VALUES (0), (1), (NULL);
SELECT c0 FROM t0 WHERE c0 IS NOT 1; -- {0} 🐞 {0, NULL} ✔
```

not being fetched. We reported this bug to the SQLite developers, who stated that it existed since 2013, classified it as critical and fixed it quickly. Even for this simple query, differential testing would have been ineffective in detecting the bug. The `CREATE TABLE` statement is specific to SQLite, since, unlike other popular DBMSs, such as PostgreSQL and MySQL, SQLite does not require the column `c0` to be assigned a column type. Furthermore, both MySQL's and PostgreSQL's `IS NOT` cannot be applied to integers; they only provide an operator `IS DISTINCT FROM`, which provides equivalent functionality. All DBMSs provide an operator `IS NOT TRUE`, which, however, has different semantics; for SQLite, it would fetch only the value `0`, and not expose the bug.

To demonstrate the generality of our approach, we implemented it for three popular and widely-used DBMSs, namely SQLite [49], MySQL [36], and PostgreSQL [40]. In total, we found 96 unique bugs, namely 64 bugs in SQLite, 24 bugs in MySQL, and 8 in PostgreSQL, demonstrating that the approach is highly effective and general. 61 of these were logic bugs found by the containment oracle. In addition, we found 32 bugs by causing DBMS-internal errors, such as database corruptions, and for 3 bugs we caused DBMS crashes (*i.e.*, `SEGFAULT`s). One of the crashes that we reported for MySQL was classified as a security vulnerability (CVE-2019-2879). 78 of the bugs were fixed by the developers, indicating that they considered our bug reports useful.

Since our method is general and applicable to all DBMSs, we expect that it will be widely adopted to detect logic bugs that have so far been overlooked. In fact, after releasing a preprint of the paper, we received a number of requests by companies as well as individual developers indicating their interest in implementing PQS to test the DBMSs that they were developing. Among these, PingCAP publicly released a PQS implementation that they have been successfully using to find bugs in TiDB. For reproducibility and to facilitate further research on this topic, we have released SQLancer at https://github.com/sqlancer/. In addition, the artifact associated with the paper contains SQLancer as well as a database of all reported bugs [44]. PQS inspired complementary follow-up work, such as NoREC and TLP, which focus on finding sub-categories of logic bugs [42, 43]. Despite this, PQS has notable limitations; it only partly validates a query's result, and cannot be used, for example, to test aggregate functions, the size of the result set, or its ordering. Furthermore, the effort required to implement the technique depends on the complexity of the operations to be tested, which can be high for complex operators or functions.

In summary, we contribute the following:

- A general and highly-effective approach to finding bugs in DBMSs termed *Pivoted Query Synthesis (PQS)*.
- An implementation of PQS in a tool named SQLancer, used to test SQLite, MySQL, and PostgreSQL.
- An evaluation of PQS, which uncovered 96 bugs.

2 Background

This section provides important background information on relational DBMSs, SQL, and the DBMSs we tested.

Database management systems. We primarily aim to test *relational* DBMSs, that is, those that are based on the *relational data model* proposed by Codd [10]. Most widely-used DBMSs, such as Oracle, Microsoft SQL, PostgreSQL, MySQL, and SQLite are based on it. A relation R in this model is a mathematical relation $R \subseteq S_1 \times S_2 \times ... \times S_n$ where S_1, S_2, ..., S_n are referred to as domains. More commonly, a relation is referred to as a *table* and a domain is referred to as a *data type*. Each tuple in this relation is referred to as a row. SQL [8], a domain-specific language that is based on relational algebra [11], is the most commonly used language to interact with the DBMSs. ANSI first standardized SQL in 1987, and it has since been developed further. In practice, however, DBMSs lack functionality described by the SQL standard and deviate from it. In this paper, we assume basic familiarity with SQL.

Test oracles. An effective *test oracle* is crucial for automatic testing approaches [21]. A test oracle assesses whether a given test case has passed. Manually written test cases encode the programmer's knowledge who thus acts as a test oracle. In this work, we are interested only in automatic test oracles, which would allow comprehensively testing a DBMS. The most successful automatic test oracle for DBMSs is based on *differential testing* [46]. Differential testing refers to a technique where a single input is passed to multiple systems that implement the same language to detect mismatching outputs, which would indicate a bug. In the context of DBMSs, the input corresponds to a database as well as a query, and the systems to multiple DBMSs—when their fetched result sets mismatch, a bug in one of the DBMS would be detected. However, SQL dialects vary significantly, making it difficult to use differential testing effectively. This is also acknowledged by industry. For example, Cockroach Labs state that they *"are unable to use Postgres as an oracle because CockroachDB has slightly different semantics and SQL support, and generating queries that execute identically on both is tricky [...]"* [22]. Furthermore, differential testing is not a *precise* oracle, as it fails to detect bugs that affect all the systems.

Table 1: The DBMSs we tested are popular, complex, and have been developed for a long time.

DBMS	Popularity Rank		LOC	Released
	DB-Engines	Stack Overflow		
SQLite	11	4	0.3M	2000
MySQL	2	1	3.8M	1995
PostgreSQL	4	2	1.4M	1996

Tested DBMSs. We focused on three popular and widely-used open-source DBMSs: SQLite, MySQL, and PostgreSQL (see Table 1). According to the DB-Engines Ranking [1] and the Stack Overflow's annual Developer Survey [38], these DBMSs are among the most popular and widely-used ones. Furthermore, the SQLite website speculates that SQLite is likely used more than all other databases combined; most mobile phones extensively use SQLite, it is used in most popular web browsers, and many embedded systems (such as television sets) [48]. All DBMSs are production-level systems, and have been maintained and developed for about 20 years.

3 Pivoted Query Synthesis

We propose *Pivoted Query Synthesis* as an automatic testing technique for detecting logic bugs in DBMSs. Our core insight is that by considering only a single row at a time, a conceptually-simple test oracle can be created that can effectively detect logic bugs. Specifically, our idea is to select a random row, to which we refer as the pivot row, from a set of tables and views in the database. Subsequently, we randomly generate a set of boolean predicates, which we then modify so that they evaluate to `TRUE` for the values of the pivot row based on an Abstract Syntax Tree (AST) interpreter. By using these expressions in `WHERE` and `JOIN` clauses of an otherwise randomly-generated query, we can ensure that the pivot row must be contained in the result set. If it is not contained, a bug has been found. Basing the approach on an AST interpreter provides us with an exact oracle. While implementing this interpreter requires moderate implementation effort for complex operators (such as regular expression operators), other challenges that a DBMS has to tackle, such as query planning, concurrent access, integrity, and persistence can be disregarded by it. Furthermore, the AST interpreter can be naively implemented without affecting the tool's performance, since it only operates on a single record, whereas the DBMS has to potentially scan through all the rows of a database to process a query.

3.1 Approach Overview

Figure 1 illustrates the detailed steps of PQS. First, we create a database with one or multiple random tables, which we fill with random data (see step ①). We ensure that each table, and randomly generated view, holds at least one row, to enable selecting a random pivot row in step ②. A pivot row is only conceptually a row, and can be composed of columns that refer to rows of multiple tables and/or views. Its purpose is to use it to derive a test case as well as a test oracle to validate the correctness of the DBMS. The pivot row shown in Figure 1 consists of both columns from table `t0` and `t1`. In the next steps, we proceed by constructing a test oracle based on the pivot row. To this end, we randomly create expressions based on the DBMS' SQL grammar and valid table column names (see step ③). We evaluate these expressions, substituting column references by the corresponding values of the pivot row. Then, we modify the expressions so that they yield `TRUE` (see step ④). We use these expressions in `WHERE` and/or `JOIN` clauses for a query that we construct (see step ⑤). We pass this query to the DBMS, which returns a result set (see step ⑥), which we expect to contain the pivot row, potentially among other rows. In a final step, we check whether the pivot row is indeed contained in the result set (see step ⑦). If it is not contained, we have likely detected a bug in the DBMS. For the next iteration, we either continue with step ② and generate new queries for a newly-selected pivot row, or continue with ① to generate a new database.

Our core idea is given by how we construct the test oracle (see steps ② to ⑦). Thus, Section 3.2 first explains how we generate queries and check for containment, assuming that the database has already been created. Section 3.3 then explains step ①, namely how we generate the tables and data. Section 3.4 provides important implementation details.

3.2 Query Generation & Checking

The core idea of our approach is to construct a query for which we anticipate that the pivot row is contained in the result set. We randomly generate expressions to be used in `WHERE` and/or `JOIN` clauses of the query, and ensure that each expression evaluates to `TRUE` for the pivot row. This subsection describes how we generate random predicates that we rectify and then use in the query (*i.e.*, steps ③ to ⑤).

Random predicate generation. In step ③, we randomly generate Abstract Syntax Trees (ASTs) up to a specified maximum depth by constructing a random expression tree based on the database's schema (*i.e.*, the column names and types). For SQLite and MySQL, SQLancer generates expressions of any type, because they provide implicit conversions to boolean. For PostgreSQL, which performs few implicit conversions, the generated root node must produce a boolean value, which we achieve by selecting one of the appropriate operators (*e.g.*, a comparison operator). Algorithm 1 illustrates how generat-

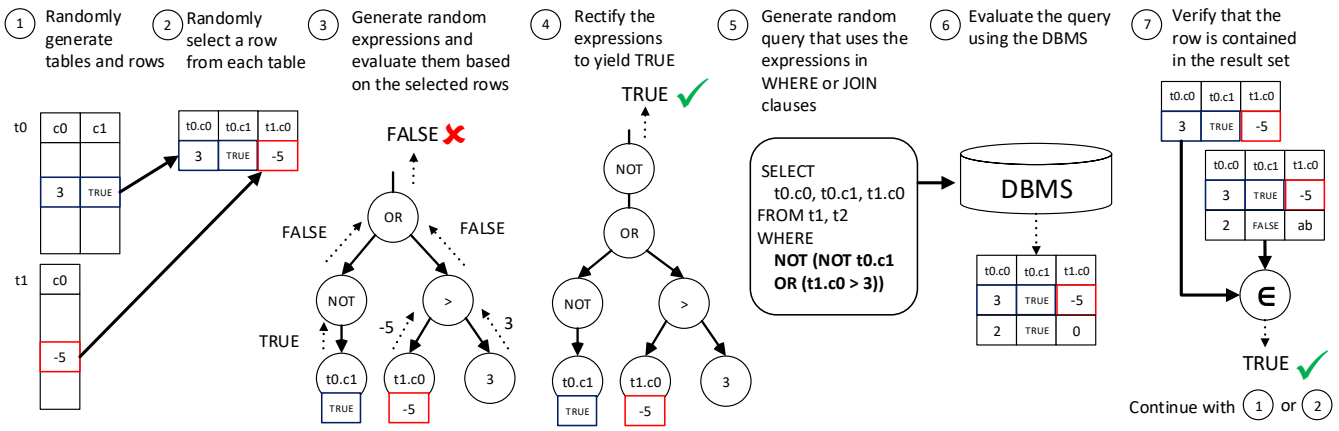


Figure 1: Overview of the approach implemented in SQLancer. Dotted lines indicate that a result is generated.

```
Function generateExpression(int depth):
    node_types ← {LITERAL, COLUMN}
    if depth < maxdepth then
        node_types ← node_types ∪ {UNARY, ...}
    type ← random(node_types)
    switch type do
        case LITERAL do
            return Literal(randomLiteral());
        case COLUMN do
            return Column-
              Value(randomTable().randomColumn());
        case UNARY do
            return
              UnaryNode(generateExpression(depth+1),
              UnaryNode.getRandomOperation());
        case ... do
            ...
    end
```

Algorithm 1: The `generateExpression()` function generates a random AST.

ing the expressions is implemented for MySQL and SQLite. The input parameter `depth` ensures that when a specified maximum depth is reached, a leaf node is generated. The leaf node can either be a randomly-generated constant, or a reference to a column in a table or view. If the maximum depth is not yet reached, also other operators are considered (*e.g.*, a unary operator such as `NOT`). Generating these expressions is dependent on which operators the respective DBMS supports. The random expression generation by itself is not a contribution of this paper; random query generators, such as RAGS [46] and SQLsmith operate similarly [45]. We implemented the expression generators manually for each DBMS under test, based on the respective DBMS SQL dialect's documentation; as part of future work, we will consider automatically deriving them based on the SQL dialect's grammar.

Expression evaluation. After building a random expression tree, we must check whether the condition yields `TRUE` for the pivot row. To this end, every node must provide an `execute()` method that computes the node's result, which needs to be manually implemented. Leaf nodes directly return their assigned constant value. Column nodes are assigned the value that corresponds to their column in the pivot row. For example, in Figure 1 step ③, the leaf node `t0.c1` returns `TRUE`, and the constant node `3` returns an integer `3`. Composite nodes compute their result based on the literals returned by their children. For example, the `NOT` node returns `FALSE`, because its child evaluates to `TRUE` (see Algorithm 2). The node first executes its subexpression, and then casts the result to a boolean; if the result is a boolean value, the value is negated; otherwise `NULL` is returned. Note that our implementation is simpler than AST interpreters for programming languages [50], since all nodes operate on literal values (*i.e.*, they do not need to consider mutable storage). It is also simpler than query engine models, such as the well-known Volcano-style iteration model [16], and widely-used models based on it, such as the vectorized model or the data-centric code generation model, which all need to consider multiple rows [26]. Since the bottleneck of our approach is the DBMS evaluating the queries rather than SQLancer, all operations are implemented naively and do not perform any optimizations. Some operations require moderate implementation effort nevertheless; for example, the implementation of the `LIKE` regular expression operator has over 50 LOC in SQLancer.

Expression rectification. After generating random expressions, step ④ ensures that they evaluate to `TRUE`. SQL is based on a three-valued logic. Thus, when evaluated in a boolean context, an expression either yields `TRUE`, `FALSE`, or `NULL`. To rectify an expression to yield `TRUE`, we use Algorithm 3. For example, in Figure 1 step ④, we modify the expression by adding a preceding `NOT`, so that the expression evaluates to `TRUE`. Note that our approach works also for other logic systems (*e.g.*, four-valued logic), by adjusting this step. Alternatively, it could be checked that the pivot row is ex-

```
Method NotNode::execute():
    value ← child.execute()
    switch asBoolean(value) do
        case TRUE do
            result ← FALSE
        case FALSE do
            result ← TRUE
        case NULL do
            result ← NULL
    end
    return result;
```

Algorithm 2: The execute() implementation of a `NOT` node.

```
Function rectifyCondition(randexpr):
    switch randexpr.execute() do
        case TRUE do
            result ← randexpr
        case FALSE do
            result ← NOT randexpr
        case NULL do
            result ← randexpr ISNULL
    end
    return result;
```

Algorithm 3: The expression rectification step applied to a randomly-generated expression.

pectedly not contained in the result set, by ensuring that the expression evaluates to `FALSE`.

Query generation. In step ⑤, we generate targeted queries that fetch the pivot row. Most importantly, the expressions evaluating to `TRUE` are used in `WHERE` clauses, which restrict which rows a query fetches, and in `JOIN` clauses, which are used to join tables. Since the expressions evaluate to `TRUE`, the pivot row is guaranteed to be contained in the result set. `JOIN` clauses are not treated specially; as we create the clause's predicate to yield `TRUE` for the pivot row, inner, full, left, and right joins all behave in the same way as a `WHERE` clause with respect to the pivot row. `SELECT` statements typically provide various keywords to control the query's behavior, from which we randomly select applicable options. Specifically, we considered the following elements:

- `DISTINCT` clauses, which filter out duplicate rows, while retaining the guarantee that the pivot row is contained in the result set;
- `GROUP BY` clauses that contain all pivot row columns to guarantee that the pivot row is contained in the result set;
- `ORDER BY` clauses, which influence only the order of the result set, which is not validated by PQS;
- aggregate functions, which compute values over multiple rows, when only a single row is present in a table, which allows partially testing them;
- DBMS-specific query options, such as the MySQL-specific `FOR UPDATE` clause, which must not influence the result set.

These additional elements are an optional extension to our core approach, and allowed PQS to find additional bugs by stressing the DBMSs' query optimizer. However, they do not comprehensively test these features.

Checking containment. After using the DBMS to evaluate the query in step ⑥, checking whether the pivot row is part of the result set is the last step of our approach. While the checking routine could have been implemented in SQLancer, we instead construct the query so that it checks for containment, effectively combining steps ⑥ and ⑦. DBMSs provide various operators to check for containment, such as the `IN` and `INTERSECT` operators. For example, for checking containment in Figure 1 step ⑦, we can check whether the row `(3, TRUE, -5)` is contained in the result set using the query shown in Listing 2, which returns a row if the pivot row is contained.

Checking arbitrary expressions. An extension of the initial idea of PQS is to use arbitrary expressions to specify which data to fetch in the query of step ⑤, rather than referring to columns only. For example, rather than referring to `t0.c0`, we might want to check whether `t0.c0 + 1` evaluates correctly. To this end, we can generalize the definition of a pivot row to refer to arbitrary computed values. For example, the pivot row value for `t0.c0 + 1` must be `4`, which can be derived based on the expression evaluation mechanism already explained for step ②. In terms of implementation, this thus requires that first the expressions to be used in step ⑤ must be generated, so that they can be evaluated to derive the pivot row values as part of step ②.

3.3 Random State Generation

In step ①, we generate a random database state. Similarly to the generation of queries, we heuristically and iteratively select a number of applicable options. The first step is fixed and consists of creating a number of tables, using the `CREATE TABLE` statement. Subsequent statements are chosen heuristically. Among the applicable options is the `INSERT` statement, which allows inserting data rows. By generating Data Definition Language as well as Data Manipulation Language statements, we can explore a larger space of databases, some of which exposed DBMS bugs. For example, we implemented `UPDATE`, `DELETE`, `ALTER TABLE`, and `CREATE INDEX` commands

Listing 2: Checking containment using the `INTERSECT` operator in SQLite.

```
SELECT (3, TRUE, -5) INTERSECT SELECT t0.c0, t0.c1
    , t1.c0 FROM t1, t2 WHERE NOT(NOT(t0.c1 OR (t1
    .c0 > 3)));
```

for all databases, as well as DBMS-specific run-time options. A number of commands that we implemented were unique to the respective DBMS. Statements unique to MySQL were `REPAIR TABLE` and `CHECK TABLE`. The statements `DISCARD` and `CREATE STATISTICS` were unique to PostgreSQL. Since the statements are chosen heuristically, the database state generation step might yield an empty database (*e.g.*, because a `DELETE` statement might have deleted all rows, or because a table constraint might make it impossible to insert any rows); in such a case, the current database is discarded and a new database is created. The random database generation is not a contribution of this paper; in fact, many database generation approaches have been proposed, any of which could be paired with PQS [5, 6, 17, 20, 27, 37].

3.4 Important Implementation Details

This section explains implementation decisions, which we consider significant for the outcome of our study.

Error handling. We attempt to generate statements that are correct both syntactically and semantically. However, generating semantically correct statements is sometimes impractical. For example, an `INSERT` might fail when a value already present in a `UNIQUE` column is inserted again; preventing such an error would require scanning every row in the respective table. Rather than checking for such cases, which would involve additional implementation effort and a run-time performance cost, we defined a list of error messages that we might expect when executing the respective statement. Often, we associated an error message to a statement depending on presence or absence of specific keywords; for example, an `INSERT OR IGNORE` is expected to ignore many error messages that would appear without the `OR IGNORE`. If the DBMS returns an expected error, it is ignored. Unexpected errors indicate bugs in the DBMS. For example, in SQLite, a *malformed database disk image* error message is always unexpected, since it indicates the corruption of the database.

Performance. We optimized SQLancer to take advantage of the underlying hardware. We parallelized the system by running each thread on a distinct database, which also resulted in bugs connected to race conditions being found. To fully utilize each CPU, we decreased the probability of SQL statements being generated that cause low CPU utilization (such as `VACUUM` in PostgreSQL). Typically, SQLancer generates 5,0000 to 20,000 statements per second, depending on the DBMS under test. Since the DBMSs we tested processed queries much faster than other statements, SQLancer generates 100,000 random queries for each database. We implemented the system in Java. However, any other programming language would have been equally well suited, as the performance bottleneck was the DBMS executing the queries.

Number of rows. We found most bugs by restricting the number of rows inserted to a low value (10–30 rows). A higher number would have caused queries to time out when tables are joined without a restrictive join clause. For example, in a query `SELECT * FROM t0, t1, t2`, the largest result set for 100 rows in each table would already be $|t0| * |t1| * |t2| = 1,000,000$, significantly lowering the query throughput. A potential concern is that this might prevent PQS from detecting bugs that are triggered only for tables with many rows. We believe that future work could tackle this by generating targeted queries for which the cardinality of the result is bounded.

Database state. For the generation of many SQL statements, knowledge of the database schema or other database state is required; for example, to insert data, SQLancer must determine the name of a table and its columns. We query such state dynamically from the DBMS, rather than tracking or computing it ourselves, which would require additional implementation effort. For example, to query the name of the tables, both MySQL and PostgreSQL provide an information table `information_schema.tables` and SQLite a table `sqlite_master`.

Bailouts. For some operators or functions, corner-case behavior (*e.g.*, how an integer operation behaves on an integer overflow) might be difficult to implement, and—at least initially—be less important to test. Unlike the DBMS, the expression evaluation step in our approach is not required to compute a result for every possible input; in our implementation, each operation can bail out during evaluation by throwing an exception, indicating that a new expression should be generated. We also use this mechanism to prevent reporting known bugs, by bailing out when input is encountered that is known to potentially trigger an already-reported bug.

Value caching. When randomly generating values, SQLancer stores values in a cache, which are subsequently re-used with a given probability. Our intuition was that this would more likely trigger interesting corner cases (*e.g.*, when comparing the same values such as $3 > 3$). Additionally, we expected this to increase the chance of successfully generating rows for tables that constraint a column to refer to another table (*i.e.*, foreign key constraints).

Implementation scope. Each testing implementation that we realized is extensive, but incomplete. For each DBMS, we implemented at least integer and string data types; for the SQLite implementation, which is the most complete one, we also support floating-point numbers and binary data. We implemented the generation of many common statements, operators, and functions. Given the size of the implementation, exhaustively enumerating all supported features is infeasible; the artifact associated with the paper can be used to investigate which features are supported. Section 5.3 gives an overview of the size of each testing implementation.

4 Evaluation

We evaluated whether the proposed approach is effective in finding bugs in DBMSs. We expected it to detect logic bugs, which cannot be found by fuzzers, rather than crash bugs. This section overviews the experimental setup, bugs found, and characterizes the SQL statements used to trigger the bugs. We then present a DBMS-specific bug overview, where we present interesting bugs and bug trends. To put these findings into context, we measured the size of SQLancer's components and the coverage it reaches on the tested DBMSs.

4.1 Experimental Setup

To test the effectiveness of our approach, we implemented SQLancer and tested SQLite, MySQL, and PostgreSQL in a period of about three months. We conducted all experiments using a laptop with a 6-core Intel i7-8850H CPU at 2.60 GHz and 32 GB of memory running Ubuntu 19.04. Typically, we enhanced SQLancer to test a new operator or DBMS feature, let the tool run for several seconds up to a day, and inspected the bugs found during this process. We automatically reduced test cases to minimal versions [41], and reduced them further manually when this helped to better demonstrate the underlying bug. Finally, we reported any new bugs found during this process. Where possible, we waited for bug fixes before continuing testing and implementing new features.

Baseline. There is no applicable baseline to which we could compare our work. RAGS [46], which was proposed more than 20 years ago, would be the closest related work, but is not publicly available. Due to the small common SQL core, we would expect that RAGS could not find most of the bugs that we found. Khalek et al. worked on automating testing DBMSs using constraint solving [3, 27], with which they found a previously unknown bug. Also their tool is not available publicly. SQLsmith [45], AFL [2] as well as other random query generators and fuzzers [39] only detect crash bugs in DBMSs. Thus, the only potential overlap between these tools and SQLancer would be the crash bugs that we found, which are not the focus of this work.

DBMS versions. For all DBMSs, we started testing the latest release version, which was SQLite 3.28, MySQL 8.0.16, and PostgreSQL 11.4. For SQLite, we switched to the latest trunk version (*i.e.*, the latest non-release version of the source code) after the first bugs were fixed. For MySQL, we also tested version 8.0.17 after it was released. For PostgreSQL, we switched to the latest beta version (PostgreSQL Beta 2) after opening duplicate bug reports. Eventually, we continued to test the latest trunk version.

Bottleneck. We found that duplicate bugs were a significant factor that slowed down our testing. After reporting a bug, we typically waited for bug fixes before continuing our bug-finding efforts; for bugs that were not quickly fixed, we attempted to avoid generating bug-inducing test cases that triggered known bugs. For SQLite, the developers reacted to most of our bug reports shortly after reporting them, and fixed issues typically within a day. Consequently, we focused our testing efforts on this DBMS. For SQLite, we also tested `VIEW`s, non-default `COLLATE`s (which define how strings are compared), floating-point support, and aggregate functions, which we omitted for the other DBMSs. For MySQL, bug reports were typically verified within a day by a tester. MySQL's development is not open to the general public. Although we tried to establish contact with MySQL developers, we could not obtain any information that went beyond what is visible on the public bug tracker. Thus, it is likely that some of the verified bug reports will subsequently be considered as duplicates or classified to work as intended. Furthermore, although MySQL is available as open-source software, only the code for the latest release version is provided, so any bug fixes could be verified only with the subsequent release. This was a significant factor that restricted us in finding bugs in MySQL; due to the increased effort of verifying whether a newly found bug was already reported, we invested limited effort into testing MySQL. For PostgreSQL, we received feedback to bug reports within a day, and it typically took multiple days or weeks until a bug was fixed, since possible fixes and patches were discussed intensively on the mailing list. As we found fewer bugs for PostgreSQL overall, the response time did not restrict our testing efforts. Note that not all confirmed bugs were fixed. For example, for one reported bug, a developer decided to "*put this on the back burner until we have some consensus how to proceed on that*"; from the discussion, we speculate that the changes needed to address the bug properly were considered too invasive.

Table 2: Total number of reported bugs and their status.

			Closed	
DBMS	Fixed	Verified	Intended	Duplicate
SQLite	64	0	4	2
MySQL	17	7	2	4
PostgreSQL	5	3	7	6

4.2 Bug Reports Overview

Table 2 shows the number of bugs that we reported (121 overall). We considered 96 bugs as true bugs, as they resulted in code fixes (78 reports), documentation fixes (8 reports), or were confirmed by the developers (10 reports). Each such bug was previously unknown and has a unique fix associated with it, or has been confirmed by the developers to be a unique bug. We opened 25 bug reports that we classified as false bugs, because behavior exhibited in the bug reports was considered to work as intended (13 reports) or because bugs that we reported were considered to be duplicates (12 reports).

Table 3: A classification of the true bugs by the bug kind.

DBMS	Logic	Error	SEGFAULT
SQLite	46	16	2
MySQL	14	9	1
PostgreSQL	1	7	0
Sum	61	32	3

Severity levels. Only for SQLite, bugs were assigned a severity level by the DBMS developers. 14 bugs were classified as *Critical*, 8 bugs as *Severe*, and 16 as *Important*. For 13 bugs, we reported them on the mailing list and no entry in the bug tracker was created. The other bug reports were assigned low severity levels such as *Minor*. While the severity level was not set consistently, this still provides evidence that we found many critical bugs.

Bug classification. Table 3 shows a classification of the true bugs. The containment oracle, which found all logic bugs, accounts for most of the bugs that we found, which is expected, since our approach mainly builds on this oracle. Perhaps surprisingly, encountering unexpected errors also allowed us to detect a large number of bugs. For PostgreSQL, we even found 7 unexpected-error bugs, while finding only 1 logic bug. We believe that this observation could be used when using fuzzers to test DBMSs, for example, by checking for specific error messages that indicate database corruptions. Our approach also detected a number of crash bugs, one of which was considered a security vulnerability in MySQL (CVE-2019-2879). These bugs are less interesting, since they could also have been found by traditional fuzzers. In fact, a duplicate bug report was reported for PostgreSQL, based on a SQLsmith finding, shortly after we found and reported it.

4.3 SQL Statements Overview

Test case length. Our automatically and manually reduced test cases—which comprise both the statements used to generate the state, as well as the bug-inducing query—typically comprised only a few SQL statements (3.71 LOC on average). For 13 test cases, a single line was sufficient. Such test cases were either `SELECT` statements that operated on constants, or operations that set DBMS-specific options. The maximum number of statements required to reproduce a bug was 8. A PostgreSQL crash bug that had already been fixed when we reported it required even 27 statements to be reproduced. Overall, the small number of statements required to reproduce a bug suggests that statements and queries could be systematically generated to efficiently, rather than randomly, explore the space (*e.g.*, such as the bounded black-box testing approach implemented in *ACE* [35]).

Statement distribution. Figure 2 shows the distribution of statements. Note that for some bug reports, we had to select the simplest test case among multiple failing ones, which might skew these results. The `CREATE TABLE` and `INSERT` statements are part of most bug reports for all DBMSs, which is expected, since only few bugs can be reproduced without manipulating or fetching data from a table. 91.0% of the bug reports included only a single table. The `SELECT` statement also ranks highly, since the containment oracle relies on it. In all DBMSs, the `CREATE INDEX` statements rank highly; especially for SQLite, we reported a number of bugs where creating an index resulted in a malformed database image or in a row not being fetched. We found that statements that compute or recompute table state were error-prone, for example, `REPAIR TABLE` and `CHECK TABLE` in MySQL, as well as `VACUUM` and `REINDEX` in SQLite and PostgreSQL. DBMS-specific options, such as `SET` in MySQL and PostgreSQL, and `PRAGMA` in SQLite also resulted in bugs being found. For PostgreSQL, some test cases contained `ANALYZE`, which gathers statistics to be used by the query planner.

Column constraints. Column constraints, which can be used to restrict the values stored in a column, were often part of test cases. The most common constraint was `UNIQUE`(appearing in 21.9% of the test cases). Also `PRIMARY KEY` columns were frequent (16.7%). Typically, the DBMSs enforce `UNIQUE` and `PRIMARY KEY` by creating indexes; explicit indexes, created by `CREATE INDEX` were more common, however (27.1%). Other constraints were uncommon, for example, `FOREIGN KEY`s appeared only in 1.0% of the bug reports.

5 Interesting Bugs

In this section, we present bugs that we found using PQS. We chose bugs that we considered to be interesting, meaning that the selection is necessarily subjective.

5.1 Containment Bugs

We consider bugs found by the containment oracle to be the most interesting, and we designed PQS to specifically find these kind of bugs.

First SQLite bug. Listing 3 shows a test case for the first bug that we found with our approach, and where SQLite failed to fetch a row. The `COLLATE NOCASE` clause instructs the DBMS to ignore the casing when comparing strings; in this test case, it unexpectedly caused the upper-case `'A'` to be omitted from the result set. The bug was classified as *Severe* and goes back to when `WITHOUT ROWID` tables were introduced in 2013. It is a typical bug that we found in SQLite, since it relies on multiple features. As with this bug, 17 of our SQLite bug reports included indexes, 11 included `COLLATE` sequences, and 5 `WITHOUT ROWID` tables.

SQLite skip-scan optimization bug. A number of SQLite bugs stem from incorrect optimizations, such as the one in

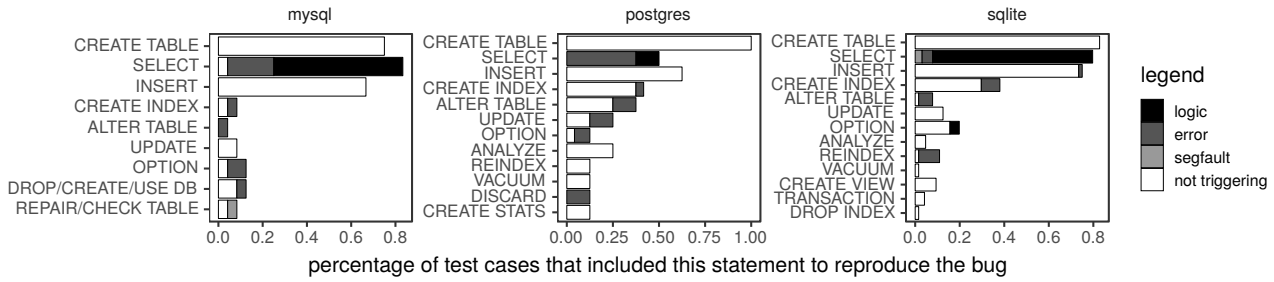


Figure 2: The distribution of the SQL statements used in the bug reports to reproduce the bug. A non-white filling indicates that a statement of the respective category triggered the bug, which was exposed by the test oracle as indicated by the filling (*i.e.*, it was the last statement in the bug report).

Listing 3: The first bug that we found with our approach involved a `COLLATE` index, and a `WITHOUT ROWID` table.

```
CREATE TABLE t0(c0 TEXT PRIMARY KEY)
       WITHOUT ROWID;
CREATE INDEX i0 ON t0(c0 COLLATE NOCASE);
INSERT INTO t0(c0) VALUES ('A');
INSERT INTO t0(c0) VALUES ('a');
SELECT * FROM t0; -- {'a'}🐞 {'A', 'a'}✔
```

Listing 4: SQLite's skip-scan optimization was implemented incorrectly for `DISTINCT`.

```
CREATE TABLE t0(c0, c1, c2, c3, PRIMARY KEY (c3,
    c2));
INSERT INTO t0(c2) VALUES (0),(0),(0),(0),(0),
  (0),(0),(0),(0),(0),(NULL),(1),(0);
UPDATE t0 SET c1 = 0;
INSERT INTO t0(c0) VALUES (0), (0), (NULL), (0),
    (0);
ANALYZE t0;
UPDATE t0 SET c2 = 1;
SELECT DISTINCT * FROM t0 WHERE c2 = 1; -- {NULL
    |0|1|NULL}🐞 {NULL|0|1|NULL, 0|NULL|1|NULL,
    NULL|NULL|1|NULL}✔
```

Listing 4. For the query in this test case, the skip-scan optimization, where an index is used even if its columns are not part of the `WHERE` clause, was implemented incorrectly for `DISTINCT` queries. The bug was classified as *Severe*.

SQLite unexpected type. Listing 5 shows a bug where an optimization for the `LIKE` operator was implemented incorrectly when applied to `INT` values. The operator was expected to fetch the row, since it checks for an exact string match, but omitted the row from the result set. While this is a minor bug, it is nevertheless interesting, considering that only SQLite allows storing a value of a type that does not match the column declaration. We found this feature to be error-prone, and discovered 8 bugs related to it.

MySQL engine-specific bug. Unlike the other DBMSs we tested, MySQL provides various engines that can be assigned to tables. Listing 6 demonstrates one bug where a row was not

Listing 5: We discovered 4 bugs in a `LIKE` optimization, one demonstrated by this test case.

```
CREATE TABLE t0(c0 INT UNIQUE COLLATE NOCASE);
INSERT INTO t0(c0) VALUES ('./');
SELECT * FROM t0 WHERE c0 LIKE './'; -- {}🐞
    {'./'}✔
```

Listing 6: We found 5 bugs using non-default engines in MySQL.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 INT) ENGINE = MEMORY;
INSERT INTO t0(c0) VALUES(0);
INSERT INTO t1(c0) VALUES(-1);
SELECT * FROM t0, t1 WHERE CAST(t1.c0 AS
  UNSIGNED) > IFNULL("u", t0.c0); -- {}🐞 {0|-1}✔
```

Listing 7: Custom comparison operator results in incorrect result.

```
CREATE TABLE t0(c0 TINYINT);
INSERT INTO t0(c0) VALUES(NULL);
SELECT * FROM t0 WHERE NOT(t0.c0 <=> 2035382037);
    -- {}🐞 {NULL}✔
```

fetched when using the `MEMORY` engine. This was one of 5 bugs that were triggered only when using a non-default engine. This test case is also interesting, as it is one of 4 MySQL test cases that relies on a cast to an unsigned integer, a type that is not provided by the other DBMSs we tested.

MySQL value range bug. We found bugs in MySQL where queries were handled incorrectly depending on the magnitude of an integer or floating-point number. For example, Listing 7 shows a bug where the MySQL-specific `<=>` inequality operator, which yields a boolean value even when an argument is `NULL`, yielded `FALSE` when the column value was compared with a constant that was greater than what the column's type can represent.

MySQL double negation bug. Listing 8 shows an interesting optimization bug that we found in MySQL. MySQL optimized away the double negation, which appears to be cor-

Listing 8: Double negation bug in MySQL.

```
CREATE TABLE t0(c0 INT);
INSERT INTO t0(c0) VALUES(1);
SELECT * FROM t0 WHERE 123 != (NOT (NOT 123)); --
    {}🐞 {1}✔
```

Listing 9: Table inheritance bug in PostgreSQL.

```
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
INSERT INTO t1(c0, c1) VALUES(0, 1);
SELECT c0, c1 FROM t0 GROUP BY c0, c1; -- {0|0}🐞
    {0|0, 0|1}✔
```

Listing 10: This bug report caused the SQLite developers to disallow double quotes in indexes.

```
CREATE TABLE t0(c0, c1);
INSERT INTO t0(c0, c1) VALUES ('a', 1);
CREATE INDEX i0 ON t0("C3");
ALTER TABLE t0 RENAME COLUMN c0 TO c3;
SELECT DISTINCT * FROM t0;--{'C3'|1}🐞{'a'|1}✔
```

rect on the first sight. However, since MySQL's flexible type system allows, for example, integers as argument to the `NOT` operator, this optimization is not generally correct. Applying `NOT` to a non-zero integer value should yield 0, and negating 0 should yield 1, which is why the predicate in the `WHERE` clause must yield `TRUE`. However, after optimizing away the double negation, the predicate effectively corresponded to `123 != 123`, which evaluated to `FALSE`, and omitted the pivot row. We considered this case as a duplicate, since the underlying bug that this test case demonstrates seems to have been fixed already in a version not released to the public. We believe that the implicit conversions provided by MySQL (and also SQLite) is one of the reasons that we found more bugs in these DBMSs than in PostgreSQL.

PostgreSQL inheritance bug. In PostgreSQL, we found only one logic bug. The bug was related to table inheritance, a feature that only PostgreSQL provides (see Listing 9). Table `t1` inherits from `t0`, and PostgreSQL merges the `c0` column in both tables. As described in the PostgreSQL documentation, `t1` does not respect the `PRIMARY KEY` restriction of `t0`. This was not considered when implementing the `GROUP BY` clause, which caused PostgreSQL to omit one row in its result set.

SQLite double quote bug. Listing 10 shows a test case, for which, after the `RENAME` operation, it is ambiguous whether the index refers to a string or column. The `SELECT` fetches `C3` as a value for the column `c3`, which is incorrect in either case. SQLite allowed both single quotes and double quotes to be used to denote strings; depending on the context, either can refer to a column name. After we reported the bug, a breaking change that disallowed strings in double quotes when creating indexes was introduced.

Listing 11: We found 4 malformed database errors in SQLite using the error oracle, such as this one.

```
CREATE TABLE t1 (c0, c1 REAL PRIMARY KEY);
INSERT INTO t1(c0, c1) VALUES (TRUE,
    9223372036854775807), (TRUE, 0);
UPDATE t1 SET c0 = NULL;
UPDATE OR REPLACE t1 SET c1 = 1;
SELECT DISTINCT * FROM t1 WHERE c0 IS NULL;--
    Error: database disk image is malformed 🐞
```

Listing 12: Unexpected null value bug in PostgreSQL.

```
CREATE TABLE t0(c0 TEXT);
INSERT INTO t0(c0) VALUES('b'), ('a');
ANALYZE;
INSERT INTO t0(c0) VALUES (NULL);
UPDATE t0 SET c0 = 'a';
CREATE INDEX i0 ON t0(c0);
SELECT * FROM t0 WHERE 'baaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaa' > t0.c0; -- Error: found
    unexpected null value in index "i0"🐞
```

5.2 Error Bugs

While finding error bugs was not the main goal of our work, they were common, which is why we discuss two such cases.

SQLite database corruption. Listing 11 shows a test case where manipulating values in a `REAL PRIMARY KEY` column resulted in a corrupted database. We found 4 such cases, as indicated by *malformed database schema* errors. This specific bug was introduced in 2015, and went undetected until we reported it in 2019; it was assigned a *Severe* severity level.

PostgreSQL multithreaded error. Listing 12 shows a bug that was triggered only when another thread opened a transaction, holding a snapshot with the `NULL` value. In order to reproduce such bugs, we had to record and replay traces of all executing threads. 4 reported PostgreSQL bugs (including closed/duplicate ones) could be reproduced only when running multiple threads.

5.3 Implementation Size and Coverage

Implementation effort. It is difficult to quantify the effort that we invested in implementing support for each DBMS, since, for example, we got more efficient in implementing support over time. The LOC of code of the individual testing components (see Table 4) reflects our estimates that we invested the most effort to test SQLite, then PostgreSQL, and then MySQL. The code part shared by the components is rather small (918 LOC), which provides evidence for the different SQL dialects that they support. We believe that the implementation effort for SQLancer is small when compared to the size of the tested DBMSs. The LOC in this table were derived after compiling the respective DBMS using default configurations, and thus include only those lines reachable in the binary. Thus, they are significantly smaller than the ones we derived statically for the entire repositories in Table 1.

Table 4: The size of SQLancer's components specific and common to the tested databases.

	LOC			Coverage	
DBMS	SQLancer	DBMS	Ratio	Line	Branch
SQLite	6,501	49,703	13.1%	43.0%	38.4%
MySQL	3,995	707,803	0.6%	24.4%	13.0%
PostgreSQL	4,981	329,999	1.5%	23.7%	16.6%

Coverage. To estimate how much code of the DBMSs we tested, we instrumented each DBMS and ran SQLancer for 24 hours (see Table 4). The coverage appears to be low (less than 50% for all DBMSs); however, this is expected, because we were only concerned about testing data-centric SQL statements. MySQL and PostgreSQL provide features such as user management, replication, and maintenance functionalities, which we did not test. Furthermore, all DBMSs provide consoles to interact with the DBMS and programming APIs. We currently do not test many data types, language elements such transaction savepoints, many DBMS-specific functions, configuration options, and operations that might conflict with other threads running on a distinct database. The coverage for SQLite is the highest, reflecting that we invested the most effort in testing it, but also that it provides fewer features in addition to its SQL implementation.

6 Discussion

Number of bugs and code quality. The number of bugs that we found in the respective DBMSs depended on many, difficult-to-quantify factors. We found most bugs in SQLite. A significant reason for this is that we focused on this DBMS, because the developers quickly fixed all bugs. Furthermore, while the SQL dialect supported by SQLite is compact, we perceived it to be the most flexible one; for example, column types are not enforced, leading to bugs that were not present in PostgreSQL, and to a lesser degree in MySQL. MySQL's release policy made it difficult to test it efficiently, limiting the number of bugs that we found in this DBMS. In PostgreSQL, we found the least number of bugs, and we believe that a significant reason for this is that the SQL dialect support is strict, and few implicit conversions are performed.

False positives. In principle, PQS does not report false positives; that is, bugs found by PQS are always real bugs. Nevertheless, false positives can be due to a limited understanding of the DBMS operator's expected behavior when implementing the operator's `execute()` method. Consequently, the 13 bug reports that were considered to work as intended were either due to (1) an incorrect implementation of an operator in PQS, or (2) a bug found by the *error* oracle where the error was expected. False bug reports allowed us to refine our implementation based on the DBMS developer's feedback. In 8 cases, bug reports also led to documentation enhancements or fixes.

Common bugs. Common bugs that we found among all DBMSs were optimization bugs (*i.e.* where a performance optimization caused correctness issues). Often, these were related to indexes created either explicitly (*i.e.* using **`CREATE INDEX`**) or implicitly (*e.g.*, using a **`UNIQUE`** constraint), as described in Section 4.3. A number of bugs were related to the handling of **`NULL`**, which seems to be difficult to reason about for DBMS developers. Most of the other bugs we found were unique to the respective DBMS.

Existing test efforts. All three DBMSs are extensively tested. For example, SQLite, for which we found most bugs, has 662 times as much test code and test scripts than source code [47]. The core is tested by three separate test harnesses. The TCL tests comprise 45K test cases, the TH3 proprietary test harness contains about 1.7 million test instances and provides 100% branch test coverage and 100% MC/DC test coverage [25], and the SQL Logic Test runs about 7.2 million queries based on over 1 GB of test data. SQLite uses various fuzzers such as a random query generator called *SQL Fuzz*, a proprietary fuzzer *dbsqlfuzz*, and it is fuzzed by Google's OSS Fuzz project [14]. Other kinds of tests are also applied, such as crash testing, to demonstrate that the database will not go corrupt on system crashes or power failures. Considering that SQLite and other DBMSs are tested this extensively, we believe that it is surprising that SQLancer could find any bugs.

Deployment. One question is how DBMS developers would use PQS during development. Similarly to fuzzers, dynamic testing approaches like PQS cannot provide any guarantees in terms of bug-finding outcomes. Consequently, it is also unclear on how long SQLancer should be run to find all bugs it would be able to find. In practice, it might be useful to run SQLancer similarly to fuzzers, for example, either for a limited period as part of an overnight continuous integration process, or constantly to maximize the chances of finding new bugs. Future work might investigate the systematic enumeration of queries, while also pruning the infinitely large space of possible queries, to give bounded guarantees.

Specification. In order to implement the expression evaluation, we implemented AST interpreters that evaluate the operators based on the pivot row. This evaluation step essentially encodes the specification against which the DBMS is checked. We implemented the expression evaluation primarily based on each DBMS' documentation. Where we deemed the documentation to be insufficient, we used a trial-and-error approach to implement the correct semantics. In contrast to differential testing, where a difference in the semantics between two DBMSs' SQL dialect would result in repeated false positives, diverging behavior in an implementation of PQS (*e.g.*, caused by implementation errors) can be addressed by code fixes. In fact, this observation can be used to effectively test the PQS implementation, by running it against the DBMS under test,

rather than—or in addition to—using manually-written unit tests.

Limitations. PQS has a number of limitations in terms of what logic bugs it can find. PQS only partly validates a query's result, and thus, in general, is inapplicable to, for example, check the correct insertion or deletion of records, detect concurrency bugs, bugs related to transactions, or bugs in the access control layer of DBMSs [19]. Conceptually, PQS cannot detect duplicate rows that are mistakenly omitted from or included in the result set, since duplicate records are indistinguishable for PQS. Consequently, it also cannot be used to validate the cardinality of a result set, even when each of its rows is once selected as a pivot row. PQS is not suited for testing the `OFFSET` and **`LIMIT`** clauses, since they might exclude the pivot row from the result set. Although PQS has found 3 bugs in aggregate functions, it can only do so in corner cases, such as when aggregate functions are used in a view that is queried, or when a table contains only a single row, in which case the result of the aggregate function can be determined easily. Similarly, PQS cannot find bugs in window functions, which also compute their result over multiple rows in a window. While SQLancer generates **`ORDER BY`** clauses, PQS cannot validate the result set's ordering. Similarly, for **`GROUP BY`** clauses, PQS cannot confirm that all duplicate values are grouped. PQS cannot be used to test **`NOT EXISTS`** predicates that reference tables (*i.e.*, semi-joins), since the approach cannot ensure that a row is not contained based on only the pivot row. Similarly, while PQS can be used to test joins, it can only test for combinations where a **`JOIN`** clauses matches rows on both the left and right side of a join; for example, for a **`LEFT JOIN`**, it is inapplicable to test cases where only values for the left table are fetched, but not the right one. PQS is unable to test the results of ambiguous queries and queries that rely on nondeterministic functions (such as used to generate random numbers), since it is based on the assumption that the result set is unambiguous. It is also unable to test user-provided functions or operators, unless they are re-implemented in PQS. Supporting these makes interesting future work. PQS, as the first practical technique for finding logic bugs in DBMSs, has demonstrated its effectiveness by finding a wide variety of bugs such as in operator implementations and optimizations.

Implementation effort. Since the supported SQL dialects differ vastly between DBMSs, we had to implement DBMS-specific components in SQLancer. It could be argued that the implementation effort is too high, especially when the full support of a SQL dialect is to be tested, which could arguably be similar to implementing a new DBMS. Indeed, we could not test complex functions such as SQLite's `printf`, which would have required significant implementation effort. However, we still argue that the implementation effort is reasonably low, and allows testing significant parts of a DBMS. Specificially, based on our experiments, implementing sargable predicates (*e.g.* those predicates for which the DBMS can use an index), already allows finding the majority of optimization bugs. Furthermore, our approach effectively evaluates only literal expressions, and does not need to consider multiple rows. This obviates the need of implementing a query planner, which typically is the most complex component of a DBMS [13]. Furthermore, the performance of the evaluation engine is insignificant; the performance bottleneck was the DBMS evaluating the queries, rather than SQLancer. Thus, we also did not implement any optimizations, which typically require much implementation effort in DBMSs [15]. Finally, we did not need to consider aspects such as concurrency and multi-user control as well as integrity [53].

7 Related Work

Testing of software systems. This paper fits into the stream of testing approaches for important software systems. Differential testing [33] is a technique that compares the results obtained by multiple systems that implement a common language; if results deviate, one or multiple of the systems are likely to have a bug. It has been used as a basis for many approaches, for example, to test C/C++ compilers [51, 52], symbolic execution engines [24], and PDF readers [30]. Metamorphic testing [9], where the program is transformed so that the same result as for the original program is expected, has been applied to various systems; for example, *equivalence modulo inputs* is a metamorphic-testing-based approach that has been used to find over one thousand bugs in widely-used compilers [31]. As another example, metamorphic testing has been successfully applied to test graphic shader compilers [12]. We present PQS as a novel approach to testing DBMSs, which solves the *oracle problem* in a novel way, namely by checking whether a DBMS works correctly for a specific query and row. We believe that our approach can also be extended to test other software systems that have an internal state, of which a single instance can be selected.

Metamorphic testing of DBMSs. PQS inspired two follow-up testing approaches, namely Non-Optimizing Reference Engine Construction (NoREC) [42] and Ternary Logic Partitioning (TLP) [43], both of which were implemented in SQLancer. Conceptually, NoREC translates a query that is potentially optimized by the DBMS (called the *optimized query*) to a query that cannot effectively be optimized, thus detecting *optimization bugs*—which are a subcategory of logic bugs—when the two query's result sets differ. TLP translates a given query to multiple so-called *partitioning queries*, each of which computes a part of the result, whose combined result is then compared with the given query's result sets. Both are metamorphic testing approaches. Thus, the effort required for implementing them is negligible; however, they cannot establish a ground truth, which PQS can. NoREC could find only 52.7% of the bugs detected by PQS, which is expected due to its narrower scope [42]. Considering that our PQS implementation could also check for non-containment, which

is a straightforward implementation enhancement, it could have detected 82.4% of the NoREC bugs. The remaining bugs found only by NoREC are due to bugs in the implementation of the `SUM()` and `COUNT()` aggregate functions, which NoREC uses for a more efficient implementation of the test oracle; it does not provide testing support for aggregates in general.

Differential testing of DBMSs. Slutz proposed an approach *RAGS* for finding bugs in DBMSs based on differential testing [46]. In *RAGS*, queries are automatically generated and evaluated by multiple DBMSs. If the results are inconsistent, a bug has been found. As acknowledged by the author, the approach was very effective, but applies to only a small set of common SQL statements. In particular, the differences in `NULL` handling, character handling, and numeric type coercions were mentioned as problematic. Our approach can detect bugs also in SQL statements unique to a DBMS, but requires separate implementations for each DBMS.

Database fuzzing. SQLsmith is a popular tool that randomly generates SQL queries to test various DBMSs [45]. SQLsmith has been highly successful and has found over 100 bugs in popular DBMSs such as PostgreSQL, SQLite and MonetDB since 2015. However, it cannot find logic bugs found by our approach. Similarly, general-purpose fuzzers such as AFL [2] are routinely applied to DBMSs, and have found many bugs, but also cannot detect logic bugs.

Consistency checking. Kingsbury has developed Jepsen, a framework to test safety properties of distributed systems (such as violations of consistency models), which found many critical bugs in distributed DBMSs [28]. As part of Jepsen, Kingsbury et al. proposed Elle [29], which is a transactional consistency checker. In contrast to PQS, Jepsen aims to find logic bugs primarily in the transaction processing of a DBMS.

Queries satisfying constraints. Some approaches improved upon random query generation by generating queries that satisfy certain constraints, such as cardinalities or coverage characteristics. The problem of generating a query, whose subexpressions must satisfy certain constraints, has been extensively studied [7, 34]; since this problem is complex, it is typically tackled by an approximate algorithm [7, 34]. An alternative approach was proposed by Bati et al. where queries are selected and mutated based on whether they increase the coverage of rarely executed code paths [4], increasing the coverage of the DBMS component under test. Rather than improved query generation, Lo et al. proposed an approach where a database is generated based on specific requirements on test queries [32]. While these approaches improve the query and database generation, they do not help in automatically finding errors, since they do not propose an approach to automatically verify the queries' results.

DBMS testing based on constraint solving. Khalek et al. worked on automating testing DBMSs using constraint solving [3, 27]. Their core idea was to use a SAT-based solver to automatically generate database data, queries, and a test oracle. In their first work, they described how to generate query-specific data to populate a database and enumerate the rows that would be fetched to construct a test oracle [27]. They could reproduce previously-reported and injected bugs, but discovered only one new bug. In follow-up work, they also demonstrated how the SAT-based approach can be used to automatically generate queries [3]. As with our approach, they provide a test oracle, and additionally a targeted data generation approach. While both approaches found bugs, our approach found many previously undiscovered bugs. Furthermore, we believe that the simplicity of our approach could make it wider applicable.

Testing other aspects. Rather than trying to improve the correctness of DBMSs, several approaches were proposed to test other aspects of DBMSs. Poess et. al proposed a template-based approach to generating queries suitable to benchmark DBMSs, which they implemented in a tool QGEN [39]. Similarly to random query generators, QGEN could also be used to test DBMSs. Gu et al presented an approach to quantify an optimizer's accuracy for a given workload by defining a metric over different execution plans for this workload, which were generated by using DBMS-specific tuning options [18]. Jung et al. found performance bugs based on several versions of a given DBMS [23]. Zheng et al. tested the ACID properties provided by the DBMS in the presence of power faults [53]. These approaches, however, cannot be used to find logic bugs.

8 Conclusion

We have presented an effective approach for detecting bugs in DBMSs, which we implemented in a tool SQLancer, with which we found over 96 bugs in three popular and widely-used DBMSs. The effectiveness of SQLancer is surprising, considering the simplicity of our approach, and that we only implemented a small subset of features that current DBMSs support. There are a number of promising directions that could help uncover additional bugs or improve PQS otherwise, which we regard as future work. SQLancer generates tables with a low number of rows to prevent timeouts of queries when multiple tables are joined with non-restrictive conditions. By generating targeted queries with conditions based on table cardinalities [7, 34], we could test the DBMSs for a large number of rows, better stressing the query planner [13]. A disadvantage of PQS is that it needs to be implemented for every DBMS to be tested. As part of future work, this effort could be reduced, for example, by providing common building blocks that could be combined to implement operators and functions more efficiently. Finally, PQS could be extended to also test for rows that are incorrectly fetched by selecting a pivot row, ensuring that the randomly-generated predicates evaluate to `FALSE` or `NULL` for it, and then check that the pivot row is not contained in the result set.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Yuanyuan Zhou, for their insightful feedback. We want to thank all the DBMS developers for responding to our bug reports as well as analyzing and fixing the bugs we reported. We especially want to thank the SQLite developers, D. Richard Hipp and Dan Kennedy, for taking all bugs we reported seriously and fixing them quickly. Furthermore, we are grateful for the feedback received by our colleagues at ETH Zurich.

References

[1] DB-Engines Ranking (December 2019), 2019. https://db-engines.com/en/ranking.

[2] american fuzzy lop, 2020. https://github.com/google/AFL.

[3] Shadi Abdul Khalek and Sarfraz Khurshid. Automated sql query generation for systematic testing of database engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 329–332, 2010.

[4] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1243–1251. VLDB Endowment, 2007.

[5] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: Generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 341–352, New York, NY, USA, 2007. Association for Computing Machinery.

[6] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 1097–1107. VLDB Endowment, 2005.

[7] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Trans. on Knowl. and Data Eng.*, 18(12):1721–1725, December 2006.

[8] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, 1974.

[9] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong, 1998.

[10] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[11] E.F. Codd. *Relational Completeness of Data Base Sublanguages*. Research report // San José Research Laboratory: Computer sciences. IBM Corporation, 1972.

[12] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA):93:1–93:29, October 2017.

[13] Leo Giakoumakis and César A Galindo-Legaria. Testing sql server's query optimizer: Challenges, techniques and experiences. *IEEE Data Eng. Bull.*, 31(1):36–43, 2008.

[14] Google. Announcing oss-fuzz: Continuous fuzzing for open source software, 2016. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[15] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[16] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, page 209–218, USA, 1993. IEEE Computer Society.

[17] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Rec.*, 23(2):243–252, May 1994.

[18] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. Testing the accuracy of query optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, DBTest '12, pages 11:1–11:6, 2012.

[19] Marco Guarnieri, Srdjan Marinovic, and David Basin. Strong and provably secure database access control. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, pages 163–178. IEEE, 2016.

[20] Kenneth Houkjær, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1243–1246. VLDB Endowment, 2006.

[21] William E. Howden. Theoretical and empirical studies of program testing. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 305–311, Piscataway, NJ, USA, 1978. IEEE Press.

[22] Matt Jibson. SQLsmith: Randomized sql testing in cockroachdb, 2019. https://www.cockroachlabs.com/blog/sqlsmith-randomized-sql-testing/.

[23] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow.*, 13(1):57–70, September 2019.

[24] Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 590–600, Piscataway, NJ, USA, 2017. IEEE Press.

[25] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.

[26] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, September 2018.

[27] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 238–247, Washington, DC, USA, 2008. IEEE Computer Society.

[28] Kyle Kingsbury. Jepsen, 2020. https://github.com/jepsen-io/jepsen.

[29] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations, 2020.

[30] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. On the correctness of electronic documents: Studying, finding, and localizing inconsistency bugs in pdf readers and files. *Empirical Softw. Engg.*, 23(6):3187–3220, December 2018.

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, 2014.

[32] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. A framework for testing dbms features. *The VLDB Journal*, 19(2):203–230, Apr 2010.

[33] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[34] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 499–510, 2008.

[35] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association.

[36] MySQL. Mysql homepage, 2020. https://www.mysql.com/.

[37] Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. Generating consistent test data: Restricting the search space by a generator formula. *The VLDB Journal*, 2(2):173–214, April 1993.

[38] Stack Overflow. Developer survey results 2019, 2019.

[39] Meikel Poess and John M. Stephens, Jr. Generating thousand benchmark queries in seconds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1045–1053. VLDB Endowment, 2004.

[40] PostgreSQL. Postgresql homepage, 2019.

[41] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. page 335–346, 2012.

[42] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, 2020.

[43] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

[44] Manuel Rigger and Zhendong Su. OSDI 20 Artifact for "Testing Database Engines via Pivoted Query Synthesis", 2020. https://doi.org/10.5281/zenodo.4005704.

[45] Andreas Seltenreich. SQLSmith, 2020. https://github.com/anse1/sqlsmith.

[46] Donald R Slutz. Massive stochastic testing of sql. In *VLDB*, volume 98, pages 618–622, 1998.

[47] SQLite. How SQLite is tested, 2020. https://www.sqlite.org/testing.html.

[48] SQLite. Most widely deployed and used database engine, 2020. https://www.sqlite.org/mostdeployed.html.

[49] SQLite. SQLite homepage, 2020. https://www.sqlite.org/.

[50] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, 2013.

[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, 2011.

[52] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 347–361, 2017.

[53] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Broomfield, CO, October 2014. USENIX Association.



Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing

Fabian Ruffy, Tao Wang, and Anirudh Sivaraman
New York University

Abstract

Programmable packet-processing devices such as programmable switches and network interface cards are becoming mainstream. These devices are configured in a domain-specific language such as P4, using a compiler to translate packet-processing programs into instructions for different targets. As networks with programmable devices become widespread, it is critical that these compilers be dependable.

This paper considers the problem of finding bugs in compilers for packet processing in the context of $P4_{16}$. We introduce domain-specific techniques to induce both abnormal termination of the compiler (crash bugs) and miscompilation (semantic bugs). We apply these techniques to (1) the open-source P4 compiler (P4C) infrastructure, which serves as a common base for different P4 back ends; (2) the P4 back end for the P4 reference software switch; and (3) the P4 back end for the Barefoot Tofino switch.

Across the 3 platforms, over 8 months of bug finding, our tool Gauntlet detected 96 new and distinct bugs (62 crash and 34 semantic), which we confirmed with the respective compiler developers. 54 have been fixed (31 crash and 23 semantic); the remaining have been assigned to a developer. Our bug-finding efforts also led to 6 P4 specification changes. We have open sourced Gauntlet at `p4gauntlet.github.io` and it now runs within P4C's continuous integration pipeline.

1 Introduction

Programmable packet-processing devices in the form of programmable switches and network interface cards (NICs) are now common. Such devices provide network flexibility, allowing network operators to customize their network, researchers to experiment with new network algorithms, and equipment vendors to upgrade features rapidly in firmware rather than waiting for new hardware. At the core of this move to programmable packet processing are the domain-specific languages (DSLs) for packet processing, along with the compilers that compile DSL programs.

Several commercial products now use such DSLs for packet processing. For instance, Intel [4], Broadcom [8], Nvidia [39], and Cisco [17] have switches and NICs programmable in DSLs such as NPL [9] and P4 [7]. Other efforts (e.g., from Google and the Open Networking Foundation (ONF)) use the P4 language to model the behavior of fixed-function devices [50].

These devices, whether fixed or programmable, are a critical part of the network infrastructure because they process every packet going through the network. Hence, a miscompiled program can persistently affect packet processing. It can also be very hard to track down miscompilations due to the lack of sophisticated debugging support on these devices. As network programmability becomes increasingly common, these DSL compilers will need to be as dependable as general-purpose compilers such as GCC and LLVM.

Motivated by these concerns, this paper considers the problem of finding bugs in compilers for packet processing. Because of the large open-source community around it, we build our work on the P4 [7] language, but our ideas also extend to similar DSLs such as NPL [9].

Bug finding in compilers is a well-studied topic, especially in the context of C [15,41,42,70,74]. Past approaches (§2) to bug finding in C compilers include fuzz testing by using randomly generated C programs [41,74], translation validation (i.e., proving that a compiler correctly translated a given input program to an output program) [48,52], and verification of individual compiler passes [45]. These prior approaches have to contend with many difficulties inherent to a general-purpose language like C, e.g., generating random programs that avoid undefined and unspecified behavior [41,74], providing semantics for pointers and memory aliasing [45], and inferring loop invariants and simulation relations to successfully perform translation validation [52].

Our key insight is that the restricted nature of a DSL such as P4 allows us to avoid much of the complexity associated with bug finding in general-purpose language compilers. In particular, the simpler nature of P4 (e.g., no loops or pointers) allowed us to more easily develop formal semantics, which

can then be used as the basis for both automated high-accuracy translation validation and model-based testing [19]. We leverage this insight to build a compiler bug-finding tool for P4 called Gauntlet. Gauntlet uses three key ideas: random program generation, translation validation, and model-based testing. We now describe these ideas and show how the restrictions of P4 allows them to be simpler than prior work.

First, we use random program generation (§4) to produce syntactically correct and well-typed P4 programs that still induce P4 compiler crashes. Because P4 has very little undefined behavior [18, §7.1.6], random program generation is considerably simpler for P4 than for C [74]. The generator does not have to painstakingly avoid generating programs with undefined and unspecified behavior, which can be interpreted differently across different compilers. The smaller and simpler grammar of P4 relative to C also simplifies the development of a random program generator.

Second, we use translation validation (§5) [48, 52] to find miscompilations in P4 compilers in which we can access the transformed program after every compiler pass. Translation validation has been used in the context of C compilers before, but has suffered one of two shortcomings. It either needs considerable manual effort per compiler pass (e.g., Crellvm [37] requires several 100 lines of manual proof-generation code for each pass; Alive [45] requires manual translation of optimizations into the Alive DSL) or suffers from a small rate of false positives and false negatives (e.g., [34, 48]). Fundamentally, this is inevitable for unrestricted C: proving program equivalence in the presence of unbounded loops is undecidable. In our case, however, the finite nature of P4[1] makes P4 program equivalence decidable and addresses both shortcomings. Thus, our use of translation validation is both precise and fully automated, requiring manual effort only to develop semantics for the P4 language—not manual effort per compiler pass.

Third, we use model-based testing (§6) to generate input-output test packets for P4 programs based on the semantics we had to develop for translation validation. We use these test packet pairs to find miscompilations in black-box and proprietary P4 compilers where we can not access the transformed program after every compiler pass. Testing for general-purpose languages [13] is effective at generating inputs that provide sufficient path coverage by finding inputs satisfying path conditions. But without language semantics, determining the correct output for these test inputs is hard. By creating formal semantics for P4 for translation validation, we are able to generate both input and output test packets, which can then be used to test the implementation produced by the compiler for a P4 program.

We applied Gauntlet to 3 platforms (§7): (1) the open-source P4 compiler infrastructure (P4C) [12], which serves as a common base for different P4 compiler implementations; (2) the P4 back end for the open-source P4 behavioral model (BMv2) [6], a reference software switch for P4; and (3) the P4 back end for Barefoot Tofino, a high-speed programmable switching chip [4]. Across these 3 platforms, and over 8 months of testing, we found a total of 96 new and distinct bugs, all of which were confirmed and assigned to a compiler developer. Our efforts also led to 6 changes [18, §A.1] to the P4 specification. 54 of these bugs have already been fixed. We analyze these bugs in detail and describe where they were found, their root causes, and which commits introduced them. Gauntlet has been merged into the continuous integration pipeline of the official P4 reference compiler [57]. Our tools are open source and available at `p4gauntlet.github.io`. To our knowledge, Gauntlet is the first example of using translation validation for compiler bug finding on a production compiler as part of its continuous integration workflow.

While Gauntlet has been very effective, it is still restricted in the kinds of bugs, compiler passes, and language constructs it can handle. We describe these restrictions to motivate future work (§8). Further, while we developed these bug-finding techniques in the context of P4, we believe the lessons we have learned (§7.4) apply beyond P4 to other DSLs with simpler semantics relative to general-purpose languages (e.g., the HLO IR for the TensorFlow [1] XLA compiler [71]).

2 Background and Motivation

2.1 Approaches to Testing Compilers

Levels of compiler testing. A compiler must reject incorrect programs with an appropriate error message and accurately translate correct programs. However, a program can be correct to varying levels. McKeeman [46] provides a taxonomy of these levels in the context of C (Table 1). Each level corresponds to the program getting deeper into the compiler before it is rejected (e.g., lexer, parser, type checker, optimizer, code generator). The difficulty of generating test programs also goes up with increasing input level. For instance, while general-purpose fuzzers such as AFL [75] are sufficient to stress test the lexer, more sophistication is required to generate syntactically correct and well-typed programs, which are required to test the optimizer. In the context of the P4 compiler, we observed very limited success in bug finding using a general-purpose fuzzer such as AFL. This is because testing at the first few levels of Table 1 is already handled adequately by P4's open-source compiler test suite [12, §3.4].

Hence, for this paper, we only consider programs at the higher levels: static, dynamic, and model-conforming. These are programs that pass the lexing, parsing, type checking, and semantic analysis phases of the compiler, but still trigger compiler bugs. Like Csmith [74], we categorize bugs into *crash bugs* and *semantic bugs*. A crash bug occurs when the compiler abnormally terminates on an input program without producing either an output program or a useful error message. Crash bugs include segmentation faults, assertion violations,

[1] Finite in that input and output packets and state are finite bit vectors. Loops are bounded (parsing [18, §12]) or forbidden (control flow [18, §13]).

Level	Input Class	Example of incorrect input
1	Sequence of ASCII characters	Binary files
2	Sequence of words and spaces	Variable name beginning with $
3	Syntactically correct	Missing semicolon
4	Type correct	Adding int to string
5	Statically conforming	Undefined variables
6	Dynamically conforming	Program throwing exceptions
7	Model-conforming	Program producing wrong outputs

Table 1: McKeeman's [46] 7 levels of C compiler correctness.

incomplete error messages, and out-of-memory errors. A semantic bug occurs when the compiler produces an output executable, but the executable's behavior is different from the input program, e.g., due to an incorrect program transformation in a compiler optimization pass. In P4, semantic bugs manifest as any packet output that differs from the expected packet output given an input packet. Crash bugs we are interested in correspond to level 5 in Table 1; semantic bugs correspond to levels 6 and 7.

Bug-finding strategies. We now look at how compiler bugs are found. A key challenge in compiler bug finding is the oracle problem. Given an input program to a compiler, the expected outcome (i.e., should it accept/reject the program and what should the output be?) is unclear unless one consults an all-knowing oracle. Below, we outline the major techniques used to approximate this oracle knowledge.

In differential testing [46], given two compilers, which both receive the same input program, if compiler A's output (after compiling and running the program) differs from compiler B's output, there is a bug in one of them. This works as long as there are at least two independent compiler implementations for the same language. Csmith [74] is one example of this approach; it feeds the same randomly generated C program to multiple C compilers and checks whether the outputs generated by executing the binary produced by each compiler differ. Another example is Different Optimization Levels (DOL) [15], which selectively omits compiler optimizations and compares compiler outputs with and without these optimization passes. If the end result differs after specific passes have been skipped or added, it points to a bug. This technique can be used in any compiler framework that supports selective omission of optimizations.

Metamorphic testing [16] can serve a similar role as differential testing, especially when multiple compilers are not readily available or optimization passes can not be easily disabled. Instead of feeding the same input program to different compilers, different input programs that are expected to produce the same compiler output are fed to the same compiler. The run-time outputs after compiling these different input programs are compared to determine if there is a bug or not. EMI is an example of this approach [41]. Given a randomly generated C program P, and random input I to this program, EMI uses the path coverage tool `gcov` [53] to identify dead code in P when run on input I. EMI then prunes away this dead code to produce new programs P' whose output must agree with P's output when run on the input I. Then EMI compiles and runs both P and P' to check whether they indeed produce the same output when given I as input.

Translation validation is a bug-finding technique that converts the program before and after a compiler optimization pass into a logical formula and checks if both programs/formulas are equivalent using a constraint solver [45, 48, 52, 76]. A failed check indicates a semantic bug. Program equivalence is an undecidable problem for Turing-complete languages such as C, requiring manual assistance to perform translation validation. Typical examples of manual assistance are (1) simulation relations, which encode correspondences between variables in two programs; and (2) loop invariants, required to prove the equivalence of programs with loops. While it is possible to just unroll loops a constant number of times [34] or learn these relations [48, 66], these techniques are not guaranteed to be precise and occasionally generate false alarms [37]. The occurrence of false alarms makes translation validation an unlikely choice for recurring use in compiler testing for general-purpose languages (e.g., for continuous integration). This is because the number of false alarms typically exceeds compiler developer tolerance.

2.2 Motivating Gauntlet's Design

Random program generation for crash bugs. From EMI and Csmith, we borrow the idea of generating random programs that are lexically, syntactically, and semantically correct. Unlike EMI and Csmith, however, our random program generation is simpler. It does not have to avoid undefined behavior, which, by design, is quite limited in $P4_{16}$. Further, generating programs with undefined behavior helps us flag compiler passes that might exploit undefined behavior in counter-intuitive ways [73]. We feed these randomly generated programs to the compiler to see if it generates a crash, typically a failure of an assertion written by the P4 compiler developers.

Translation validation for semantic bugs. Differential and metamorphic testing allow us to compare different run-time outputs from compiled programs to detect semantic bugs. However, we can not directly apply either to P4 compilers. Differential testing requires two or more independent compiler implementations that are comparable in their output. $P4_{16}$ compilers for different hardware and software targets are not comparable because program behavior is target-dependent [12, §2.1]. Presently there aren't multiple independent compilers for the same target. Developing an entirely new compiler exclusively for the sake of testing the existing compiler is not productive because it can only be reused for one target. Metamorphic testing [41], on the other hand, requires the use of code-coverage tools such as `gcov` to determine which parts of the program are touched by a given input. Concurrent research [40] has proposed such tools for

P4, but these tools were not available when we commenced work on Gauntlet.

On the other hand, P4's domain-specific restrictions make translation validation easier relative to general-purpose languages such as C. P4 programs are finite-state and finite-time, which makes program equivalence decidable at a theoretical level. At the practical level, P4's lack of pointers, memory aliasing, and unstructured control flow (e.g., goto) allow for easier generation of language semantics. Furthermore, using an SMT solver together with translation validation is more precise than randomized testing approaches such as EMI and Csmith because the solver exhaustively searches over all packet inputs to a program to find semantic bugs.

To perform translation validation, we convert P4 programs before and after a compiler pass into logic formulas and assert equivalence of these formulas. To do so, we could have converted P4 programs into C code and then asserted equality using Klee's equivalence-checking mode [13]. However, instead, we directly converted P4 programs into logic formulas in Z3 [20] for two reasons. First, the effort to convert P4 to semantically equivalent C is about the same as producing Z3 formulas directly. The difficulty lies in correctly formalizing all the language constructs of P4, not in the output format. Second, generating Z3 formulas directly gives us more control and allows us to leverage domain-specific techniques to optimize these formulas.

Model-based testing for black-box compilers. Some industry compilers do not have an open specification of their internal program representation or machine code format. In such cases, we cannot use our translation validation technique because it relies on comparing semantics before and after the compiler has transformed the program. Instead, we reuse the semantics we have generated for the input P4 program to determine test cases (i.e., input-output packet pairs) for these random programs. These test cases are then used to directly check the implementations of the P4 programs produced by these compilers. This is effectively model-based testing [19], with the Z3 semantics serving as a model of the P4 program and the compiler-generated binary being the entity under test.

2.3 Goals and Non-Goals

Find many, but not all bugs. Our goal is to find many crash and semantic bugs in the P4 compiler, but our tool is not exhaustive. Specifically, we do not intend to build or replace a fully verified compiler like CompCert [43], given the large labor and time cost associated with such an undertaking with respect to the breadth of P4 back ends. We want to strengthen existing P4 compilers, not write a safe replacement.

Check the compiler, not the programmer. We are not verifying that a particular P4 program is devoid of certain kinds of bugs. This problem is addressed by orthogonal work on P4 program verification [22, 25, 32, 44, 68] and P4 testing [67]. Although Gauntlet can in principle be used in for verifying a P4 program, we have not designed it for such use cases. The random programs we generate to find bugs in the P4 compiler are much smaller and more targeted than a typical P4 switch program. Our tool does not need to be able to generate and efficiently solve Z3 formulas for large P4 programs to tease out compiler bugs, although it achieves acceptable performance on large programs (Table 4).

Unlike p4v [44] and Vera [68], whose goal is to provide semantics to find bugs in large programs such as `switch.p4`, we have developed our semantics for efficient equality checks of diverse, but relatively small, P4 programs. Because of this difference in goals, we believe our semantics cover a broader set of P4 language constructs and corner cases than p4v and Vera—broad enough that we have found bugs in the P4 specification.

Develop target-independent techniques. We designed our tools to be as target-independent as possible and specialize them to test the front and mid end of the compiler. While we support restricted forms of back-end testing (§6), we do so in a way that allows us to quickly integrate and adapt to new back ends without having to understand detailed target-specific behavior. In particular, we do not cover target-specific semantics such as externs [18, §4.3]. We do this by generating programs that are defined in a target-neutral manner with respect to $P4_{16}$'s semantics, i.e., we avoid generating target-specific extern calls.

Only test mature compilers. We only test mature compilers such as P4C and the corresponding behavioral model[2] as well as the commercial Tofino compiler. For example, P4C supports other back ends such as the eBPF, uBPF, and PSA targets, which are pre-alpha quality and preliminary compiler toolchains. Finding bugs is likely unhelpful for the respective compiler developers at this moment.

3 Background on P4

P4 is a statically typed DSL designed to describe computations on network packet headers. This paper focuses on $P4_{16}$, the latest version of P4 [18]. Figure 1 shows the main $P4_{16}$ concepts, explained below.

Packages and targets. A P4 program consists of a set of procedures; each procedure is loaded into a programmable block of the target (e.g., a switch [4] or NIC [51]). These programmable blocks correspond to various subsystems such as the parser or the match-action pipeline. The *package* lists the available programmable blocks in a target. One example of a package for a target is the `v1model`, which models the architecture of a particular BMv2 [6] software switch target, referred to as "simple switch" [26]. For simplicity, we will

[2] Both have entered "permanent beta-status" since November 2019: https://github.com/p4lang/p4c/issues/2080

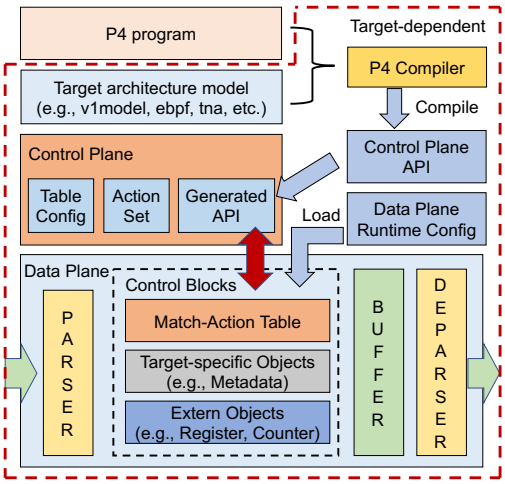


Figure 1: An example P4 compilation model.

refer to BMv2 as the target instead of simple switch.

P4 compilers. A $P4_{16}$ compiler translates a $P4_{16}$ program and the target package model into target-dependent instructions. These target instructions are combined with the non-programmable blocks (e.g., a fixed scheduler) to form the target's data plane. These instructions also specify how this data plane can be accessed and configured by the control plane (Figure 1). P4C [12] is the official open-source reference compiler infrastructure of the $P4_{16}$ language and implements the current state of the specification. P4C employs a nanopass design [65]: a composable library of front- and mid-end compiler passes that perform code analysis, transformation, and optimization on input programs. We analyze these nanopasses using translation validation.

Compiler back ends. To implement a $P4_{16}$ compiler, developers write P4C back ends, which use P4C's front- and mid-end passes along with their own back-end specific transformations, to translate $P4_{16}$ code at the conclusion of the mid end into instructions for their own target. In this paper, we focus on 2 production-grade P4C back ends: the Tofino [4] and BMv2 [6] back ends.

Parsers and control blocks. A P4 parser is a finite state machine that transforms an incoming byte sequence received at the target into a structured representation of header definitions. For example, incoming bytes may be parsed as packets containing Ethernet, IP, and TCP/UDP headers. A deparser converts this representation back into a byte sequence. Control blocks describe the per-packet operations that are performed on the input header. These operations are expressed in the form of the core primitives of the language: tables, actions, metadata, and extern objects.

Tables. Tables are objects in the control block similar to a Python dictionary. Table entries are match-action pairs inserted by the network's control plane [14, 47]. When a table is applied to a packet traversing the control block, its header is compared against the match key of all match-action entries in the table. If any entry's key matches the header, the action associated with the match is executed. Actions are procedures that can modify state and/or input headers.

Calling conventions. $P4_{16}$ uses "copy-in/copy-out" [18, §6.7] semantics for method calls. For any callable object in P4, the parameter direction (also known as mode [36, §8.2]) explicitly specifies which parameters are read-only and which parameters can be modified, with the modifications persisting after function termination. Modifiable parameters are labelled with the direction `inout` or `out` in the definition of the procedure. Read-only values are marked `in`. At the start of a procedure call, the arguments are copied left-to-right into the associated parameter slots. Parameters with `out` label remain uninitialized. Once the procedure has terminated, all procedure parameters with the label `inout` or `out` are copied back towards the original input arguments.

Metadata. Metadata is programmer-defined or target-specific data that is associated with a packet header, while it traverses the target. Examples of metadata include the packet input port, packet length, queue depth, or priority; this information is interpreted by the target according to target-specific rules. Metadata can also be modified during the execution of the control block.

Externs. Externs are an extensibility mechanism, which allows targets to describe built-in functionality. Externs are object-like and have methods. Examples include calls to checksum units, hash units, counters, and meters. P4's "copy-in/copy-out" semantics allow reasoning about externs to some degree; we can discern which input arguments can take on an arbitrary value and which arguments are read-only.

4 Random Program Generation

Gauntlet's random program generator produces valid $P4_{16}$ programs to directly trigger a crash bug. If these programs do not cause a compiler crash they serve as input for our translation validation and model-based testing techniques.

4.1 Design

We require diverse input programs to exercise code paths within many compiler passes—and hence bugs in those passes. P4C already contains a sample of over 600 programs as part of its test suite. During testing, the reference outputs of each of the test programs are textually compared to the actual outputs after the front- and mid-end passes to check for regressions [12, §3.4]. However, this comparison technique is inadequate for semantic bugs. Further, these programs are typically used to test the lexer and parser, not deeper portions of the compiler.

P4Fuzz [2] is a tool that can generate random P4 programs. However, when we tried using P4Fuzz, we found that the programs generated by it are not complex enough to find a large number of new crash or semantic bugs. For example, P4Fuzz generates programs with complex declarations (e.g., `structs` within `structs`), but does not generate programs with sufficiently complicated control flow. Hence, it does not cause P4C to execute a diverse set of compiler passes. We developed our own generator for random P4 programs that works by generating random abstract syntax trees (ASTs). With this generator we can exercise the majority of language constructs in P4. This leads to diverse test programs covering many combinations of P4 expressions. We can use these test programs to find programs that lead to unexpected crashes.

Gauntlet's random program generator is influenced by Csmith [74] and follows its philosophy of generating only well-formed input programs that pass the lexer, parser, and type checker. The generator grows an AST corresponding to the random program by probabilistically determining what kind of AST node to add to the AST at each step. By adjusting the probabilities of generating each AST node, we can steer the generator towards the language constructs we want to focus on. We can also use these probabilities to keep the size of the average generated program small, in both the number of code lines as well as program paths. With this technique we can find an ample number of semantic bugs while also avoiding programs with too many paths; such "branch" programs pose challenges for translation validation and model-based testing.

Undefined behavior. We differ from Csmith in the treatment of undefined behavior. Whereas CSmith tries to avoid generating expressions that lead to undefined behavior, we accommodate such language constructs (e.g., reading from variables that are not initialized). We record the output affected by undefined behavior as part of the logic formulas that we generate from P4 programs during translation validation (§5.2). These formulas allow us to track changes in programs with undefined behavior across compiler passes, which we use to inform compiler developers of suspicious—but not necessarily incorrect—compiler transformations [73].

4.2 Implementation

We implement our random P4 program generator as extension to P4C. The generator uses the intermediate representation (IR) of P4C to automatically grow an abstract syntax tree (AST) by expanding branches of the tree at random. For example, a block statement may generate up to (say) 10 statements or declarations, which in turn may result in further sub nodes. The generated IR AST is then converted into a P4 program using P4C's `ToP4` module. Our random program generator can be specialized towards different compiler back ends by providing a skeleton of the back-end-specific P4 package, back-end-specific restrictions, and which package blocks

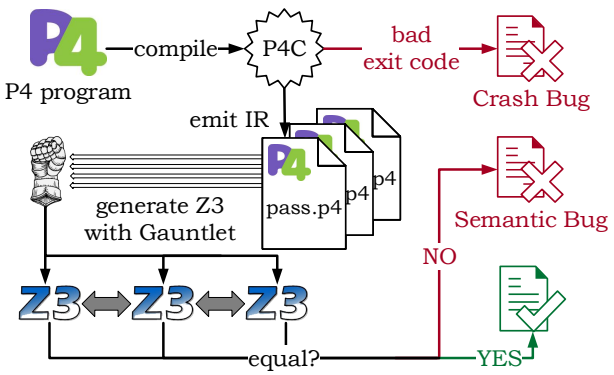


Figure 2: Translation validation in Gauntlet.

are to be filled in with randomly generated program snippets. We have currently implemented two back ends for our random program generator corresponding to the BMv2 [26] and Tofino [4] targets.

Programs generated by our random program generator are required to be syntactically sound and well-typed. Our aim is not to test if P4C can correctly catch syntax and type errors (levels 3 and 4 of Table 1). If P4C's parser and type checker (correctly) reject a generated program, we consider this to be a bug in our random program generator. For example, if an action parameter has a `inout` or `out` qualifier, only writable variables may be passed as arguments.

5 Translation Validation

To detect semantic bugs, we employ translation validation [52], a classic technique from the compiler literature in which an external tool certifies that a particular compiler pass has correctly transformed a given input program.

5.1 Design

To perform translation validation for P4, we developed a symbolic interpreter for the $P4_{16}$ language to transform P4 programs into Z3 formulas [20]. Figure 2 describes our workflow. To validate a P4 program, the symbolic interpreter converts the program into a Z3 formula capturing its input-output semantics. An equivalence checker then submits the Z3 formulas of a program before and after a compiler pass to the Z3 SMT solver. The solver tries to find an input that violates equivalence of these two formulas. If it finds such an input, this is a semantic bug. Translation validation has two advantages over random testing. First, it can accurately detect subtle differences in program semantics without any knowledge about expected input packets or table entries. Second, when we can access intermediate P4 programs after each compiler pass, we can pinpoint the erroneous pass.

5.2 Implementation

Like our random program generator, we wrote the interpreter as an extension to P4C. We use the IR generated by the P4C parser to determine the semantics of a P4 program. Each programmable block of a P4 package represents an independent Z3 formula. For example, the `v1model` package [26] of the BMv2 back end has 6 different independent programmable blocks: `Parser`, `VerifyChecksum`, `Ingress`, `Egress`, `ComputeChecksum`, and `Deparser`. For each block, we generate a separate Z3 formula.

Developing the symbolic interpreter. Overall, it took us 5 months of implementation effort until our symbolic interpreter was reliable enough to find new semantic bugs in P4 compilers, instead of encountering false alarms that were actually interpreter bugs. The fact that P4C contains a sizeable test suite [12, §3.4] was helpful in stress testing our interpreter during development. We started our development process by performing translation validation on programs in the P4C test suite. A semantic bug on one of these test programs is probably a false alarm and a bug in our interpreter. This is because it is unlikely that the compiler miscompiles test suite programs. The reference outputs of each test after the front- and mid-end passes are tracked as part of regression testing, and the reference outputs themselves are audited by the compiler developers. We also continuously consulted with the compiler developers to ensure our understanding of the language semantics was correct.

However, we quickly realized that we also needed to generate random programs to achieve coverage and truly stress test our symbolic interpreter. Subsequently, we co-evolved the interpreter with our generator. We attribute part of our success in finding bugs to this development technique, since it forced us to consider many edge cases—more than P4C does. The test suite for our interpreter now has over 600 P4C tests plus over 100 of our own tests.

Eventually, our interpreter had become complete and trustworthy enough to perform translation validation for randomly generated programs so as to trigger semantic bugs in P4C. After we had detected the first semantic bug, we randomly generated around 10000 programs every week and added the resulting compiler bugs to our backlog. Adding support for new P4 language features as part of random program generation typically first led to a crash in our interpreter. After we fixed our own interpreter, we were frequently able to find new semantic bugs in the P4 compiler that pertained to those language features. Because any of the compiler passes may have bugs, our symbolic interpreter does not rely on any compiler pass of P4C. It only relies on the P4C parser and the `ToP4` module to produce P4 code from the IR. Hence, we designed our interpreter to handle any P4 program that successfully passed the P4C parser, i.e., before the program is desugared into any normalized form. This allows us to detect semantic bugs in the earliest front-end passes.

```
struct Hdr { bit<8> a; bit<8> b; }

control ingress(inout Hdr hdr) {
  action assign() { hdr.a = 1; }
  table t {
    key = hdr.a : exact;
    actions = {
      assign();
      NoAction();
    }
    default_action = NoAction();
  }
  apply {
    t.apply();
  }
}
```

(a) Simplified P4 program applying a table.

```
Input:  t_table_key, t_action, hdr
Output: hdr_out

hdr_out =
    if (hdr.a == t_table_key) :
        if (1 == t_action) : Hdr(1, hdr.b)
        otherwise : Hdr(hdr.a, hdr.b)
    otherwise : Hdr(hdr.a, hdr.b)
```

(b) Its semantic interpretation in Z3 shown in functional form.

Figure 3: A P4 table converted to Z3 semantics.

Converting P4 programs into Z3 formulas. We now describe briefly how we convert a P4 program into a Z3 logic formula. Figure 3 shows an example. Conceptually, our goal is to represent P4 programs in a functional form so that the input-output behavior of the functional form is identical to the input-output behavior of the P4 program. To determine function inputs and outputs, we use the parameter directions of each P4 package. Parameters with the direction `inout` and `out` make up the output Z3 data type of the function whereas parameters with the `in` and `inout` are free Z3 variables that represent the input of the function.

To determine the functional form, the symbolic interpreter traverses each path through the P4 program, maintaining expressions representing path conditions for branching. Once it reaches a portion of the program where execution ends, it stores an if-then-else Z3 expression with the condition set to the path condition and the return value set to a tuple consisting of the `inout` and `out` parameters at that point. Ultimately, the interpreter will return a single nested if-then-else Z3 expression, with each branch corresponding to a unique output from the program under a set of conditions. Using this expression we can perform operations such as equivalence checking between two Z3 formulas for translation validation or querying Z3 to provide an output for particular input for test case generation.

Handling tables. The contents of a table are unknown at compile time. Since we want to make sure we cover any possible table content, we interpret match-action pairs in tables symbolically. Figure 3 describes a simplified exam-

ple of how Gauntlet interprets tables within a control block. Per match-action table call, we generate one symbolic match (`t_table_key`) and one symbolic action variable (`t_action`), which represent a single match key and its choice of action respectively. We compare the symbolic packet header with the symbolic match key (`hdr.a == t_table_key`). If the expression evaluates to true it implies the execution of a specific action, which is chosen based on the value of the symbolic action index (`t_action`). We express this as a series of nested if-then-else statements per action available to the table. Finally, if the key does not match, the default action is selected. For instance, in Figure 3, we execute action `assign` (action id 1) iff the symbolic match variable (`t_table_key`) equals the symbolic header (`hdr.a`) and the symbolic action variable (`t_action`) equals 1. With this encoding we can avoid having to use a separate symbolic match-action pair for every entry in the match-action table, which is a prohibitively large number of symbolic variables.

Header validity. The $P4_{16}$ specification does not explicitly restrict the behavior of header validity. We model our semantics to align with the implementation in P4C. We clarified these assumptions with the compiler and specification maintainers [62]. If a previously invalid header is marked valid, all fields in that header are initially undefined. If an invalid header is returned in the final output, all fields in the header are set to invalid as well.

Interpreting function calls. Any `out` parameter in a function call is initially set undefined. If the function returns, we also generate a new free Z3 variable. In our interpreter, externs are treated as a function call that returns an arbitrary value. In addition, each argument for a parameter that has the label `inout` and `out` is set to a new free Z3 variable because the behavior of extern is unknown. Copy-in/copy-out semantics, albeit necessary to control side effects in extern objects, have been a persistent source of bugs in the compiler. A significant portion of the semantic bugs we identified were caused by erroneous passes that perform incorrect argument evaluation and side effect ordering in relation to copy-in/copy-out.

Checking equivalence between P4 programs. We use `p4test` to emit a P4 program after each compiler pass. `p4test` is a P4C back end used to test P4C. It does not produce any executable output but exercises all the default front- and mid-end passes. We only examine passes that actually modify the input program and ignore any emitted intermediate program that has a hash identical to its predecessor. We explicitly reparse each emitted P4 file to also catch bugs in the parser and the `ToP4` module.

For an input program A and the transformed output program B after a compiler pass we perform a pair-wise equivalence check for each programmable block. We use our interpreter to retrieve the Z3 formulas for all programmable blocks of the program package and compare each individual block of A to the corresponding block in B. The query for the Z3 solver is a simple inequality. It is satisfiable only if there is a Z3 assignment (e.g., a packet header input or table match-action entry) in which the Z3 formula of A produces a different output from B.

If the inequality query is satisfiable, it produces the assignment that would lead to different results and saves the failed passes for later analysis. With this technique we can precisely pinpoint in which pass a semantic bug may have happened and we can also infer the packet values we need to trigger the bug. If the report turns out to be a false alarm and is not confirmed by compiler developers, this is a bug in our symbolic interpreter, which we fix. The generated Z3 formulas could in principle be very large and checking could take a long time. However, we use quantifier free formulas for the equality check, which can be solved efficiently in Z3 [20]. Even very large expression trees can be compared under a second.

Handling undefined behavior. We track changes in undefined behavior in which the undefined portion of a P4 program has more restricted (less undefined) behavior after a compiler pass. This means we can identify scenarios where the compiler transforms a program fragment based on undefined behavior. While not immediately harmful, such changes might still indicate problematic behavior in the compiler that may be surprising to a programmer [73].

To track undefined behavior, any time a variable is affected by undefined behavior (e.g., a header is set to invalid and then valid) we label that variable "undefined." This undefined variable effectively acts as taint. Every read or write to this undefined variable is tainted. When comparing Z3 formulas before and after a pass, we can choose to replace tainted expressions with concrete values in the formula before a pass.[3] With this, we can determine if a translation validation failure was caused by undefined behavior. If we find a failure based on undefined behavior, we classify it as unstable code [73] to avoid confusion with real bugs.

6 Model-Based Testing

Our approach to translation validation is applicable only in scenarios where we have access to the P4 IR (and hence the P4 program). This is because it rests on having semantics for P4. This is the case for P4C, which has a flag that allows us to emit the input P4 program after every compiler pass as a transformed P4 program [12, §3.3]. However, in the back end, a P4 compiler employs back-end-specific passes that translate P4 into proprietary formats. These formats are undocumented, making it hard to provide semantics for them. Hence, to find back-end bugs, we developed a bug-finding approach based on model-based testing [19].

[3] We only replace tainted expressions in the "before" formula so that we can detect compiler bugs where a previously well-defined expression turns undefined, which is an actual compiler bug, not just an unsafe optimization.

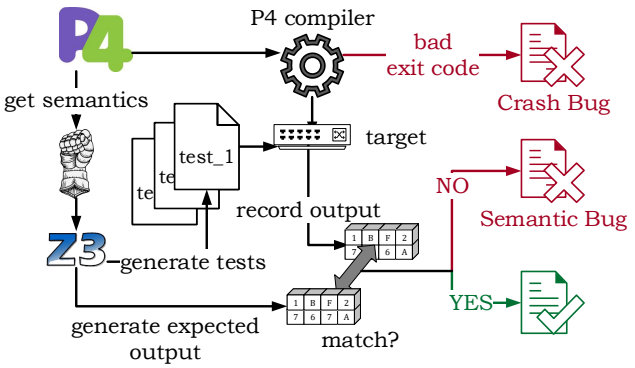


Figure 4: Model-based testing in Gauntlet.

6.1 Design

In this approach, we reuse our symbolic interpreter to produce a Z3 formula of a randomly generated P4 program (Figure 4). With this Z3 formula, we can produce input packets that traverse unique paths in the P4 program, by generating a path condition for every unique program path and asking Z3 for an input packet that satisfies this path condition. Using the same Z3 formula, we can also determine the output packet for this input packet. Thus, we generate a test case for each path and feed this case into the testing framework of the compiler's target. If the framework reports a mismatch, we know that there is likely a bug. This test technique can identify semantic bugs without requiring access to the P4 program after every intermediate compiler pass. However, unlike the translation validation approach, it is harder to pinpoint the pass causing the bug. This is effectively model-based testing [19] with the Z3 formulas being the model and the compiler output being the system under test.

6.2 Implementation

Model-based testing requires a back-end testing framework that is capable of taking input packets and producing output packets, which can then be matched against the expected output from Z3. We test two back ends: (1) the BMv2 back end that uses the simple test framework (STF) [10], which feeds packets to a software test switch and records output packet capture files and (2) the Tofino back end that uses the Packet Test Framework (PTF) [5] to inject and receive packets. We use the Tofino software simulator to check for semantic bugs in Tofino. We initially reconfirmed every semantic bug we found on the Tofino hardware target, but ultimately switched to running only the simulator for faster testing. However, we confirmed all Tofino bugs with the Tofino compiler developers.

Undefined variables. Variables affected by undefined behavior (undefined variables) are difficult to model in model-based-testing because any back end is free to perform arbitrary operations on these variables. We were left with two choices: (1) we could avoid undefined behavior in our P4 programs; (2) alternatively, we could ascribe specific values to undefined variables and check if these values conform with the implementation of the particular target. We picked the second approach because it allows independent testing of compiler optimizations in the face of undefined language constructs.

Computing input and output for test cases. We do not have control over program paths that involve undefined variables because we cannot force a target to assign specific values to such variables. Hence, we add conditions which will cause Z3 to only give us solutions for specific restricted program paths. For any path we can control (e.g., a branch that depends on the value of an input header) we compute all the possible input-output values that lead to a new path through the P4 program. This technique is computationally expensive because the number of paths can be exponential in the length of the program. However, in practice, because our P4 programs have a relatively small number of branches, test-case generation followed by testing on a P4 program still completes quickly. If members of an output header are undefined we mark those bits as "don't care" and ignore that portion of the output packet. For any invalid header we omit its member bits from the expected test output.

For every path, we feed path conditions into Z3 and retrieve a candidate set of input-output values that would cause program execution to go down that path. Because there are typically many solutions for these input-output values, we configure the Z3 solver to give us a randomized, non-zero input and its corresponding output value. In some back ends, using zero values by default may mask erroneous behavior. For example, since BMv2 initializes any undefined variable with zero, the bug in program 5c would not have been caught, had we not asked Z3 for a non-zero input-output pair.

6.3 Limitations

In contrast to translation validation that runs entirely on a formal logic-based representation of the P4 program, model-based testing has several limitations that are caused by needing to run actual end-to-end tests on real targets.

Dropped packets in the testing framework. A key assumption in the model-based-testing approach is that the generated test cases can actually be fed to the testing framework of the back end. However, the semantics of the generated P4 program do not describe hardware-specific restrictions. For example, some devices impose minimum packet size requirements or drop packets with invalid MAC addresses. More generally, we have found that test cases where the input packets have certain values in their headers can be dropped silently by the back end without generating an output packet. Effectively, there is a mismatch between the Z3 semantics, which

Bug Type	Status	P4C	BMv2	Tofino
Crash	Filed	36	4	35
	Confirmed	33	4	25
	Fixed	27	4	8
Semantic	Filed	31	1	10
	Confirmed	26	1	7
	Fixed	22	1	0
Total	**96**	**59**	**5**	**32**

Table 2: Bug summary. Unfixed bugs have been assigned.

says that a certain output packet must be produced and the back end's semantics, which produces no output packet. In such cases, we have had to discard these test cases, reducing the breadth of coverage for testing the compiler.

Unknown interfaces between programmable blocks. P4 also does not provide semantics on the treatment of packets in-between the individual control or parser blocks. This is not an issue for translation validation since we compare each programmable block individually. For an end-to-end test, however, we need to know how data is modified between these blocks so that we know what output packet to expect.

Test case complexity. Paths with many branches can generate a large number of distinct path conditions. Thus, millions of input-output packet pairs might be generated. Since small programs have sufficed so far for bug finding, we have not run into these issues. In the future, we may need an efficient path selection technique to tease out more complex bugs on closed-source compilers.

7 Results

We now analyze the P4 compiler bugs found by Gauntlet. A detailed breakdown can be found at `p4gauntlet.github.io`. Our main findings are summarized below.

1. We confirmed a total of 96 new, distinct bugs across the P4C framework and the BMv2 and Tofino P4 compilers. Of these bugs, 62 are crash and 34 are semantic bugs.
2. Our efforts led to 6 P4 specification changes [18, §A.1].
3. We achieved this in the span of only 8 months of testing with Gauntlet, and despite only generating random programs from a subset of the $P4_{16}$ language.
4. Model-based testing is effective enough to find semantic bugs in closed-source back ends such as the Tofino compiler, despite us not having access to the internal IR.

7.1 Sources of Bugs

We distinguish the bugs we found into three primary sources: bugs we found in the common P4C framework and bugs we found in the compiler back ends for BMv2 and Tofino. Both

Location	P4C	BMv2	Tofino	Total
Front End	38	-	-	38
Mid End	21	-	-	21
Back End	-	5	32	37
Total	59	5	32	**96**

Table 3: Distribution of bugs in the P4 compilers.

the BMv2 and Tofino back ends use the P4C front- and mid-end passes. Hence, most bugs detected in P4C also likely apply to these back ends. Note that since the Tofino back end is closed source, we don't know which P4C passes it uses.

All semantic bugs in P4C were found by translation validation because we had full access to the compiler IR. Where applicable, we reproduced the semantic bugs using model-based testing and attached the failing input-output packet pair with our bug report. All the semantic bugs in the Tofino compiler were found with model-based testing.

Distribution of Bugs. Table 3 lists where we identified bugs. The overall majority of bugs were found in the P4C front- and mid-end framework, mainly because we concentrated on these areas. The majority of the back end bugs were found in the Tofino compiler. There are two reasons for this. First, the Tofino back end is more complex than BMv2 as it compiles for a high-speed hardware target. Second, we did not test the BMv2 back end as extensively as other parts of the compiler.

Bugs in the P4C infrastructure. As Table 2 shows, we were able to confirm 96 distinct bugs. 59 were uncovered in P4C, with a comparable distribution of crash bugs (33) and semantic bugs (26). Initially, the majority of bugs that we found were crash bugs. However, after these crash bugs were fixed, and as our symbolic interpreter became reliable, the semantic bugs began to exceed the crash bugs.

In addition, 6 of the bugs we found led to corresponding changes in the specification as they uncovered missing cases or ambiguous behavior because our interpretation of a specific language construct clashed with the interpretation of the compiler developers and language designers. We also continuously checked out the master branch to test the latest compiler improvements for bugs. Many bugs (16 out of 59) were caused after recent merges of pull requests during the months in which we used Gauntlet for testing. Gauntlet was able to quickly detect these bugs. To catch such bugs as soon they are introduced, the P4C developers have now integrated Gauntlet into P4C's continuous integration (CI) pipeline.

Bugs in the Tofino compiler. Model-based testing on the Tofino compiler was also successful. We confirmed 25 crash bugs and 7 semantic bugs in the Tofino compiler. These bugs are all distinct from the bugs reported to P4C. The majority of bugs present in P4C could be reproduced in the Tofino compiler as well, because it uses P4C for its front and mid end.

Program	Arch	LoC	Time (mm:ss)
tna_simple_switch.p4	TNA	1940	**00:05**
switch_tofino_x0.p4	TNA	5751	**00:51**
switch_tofino2_y0.p4	TNA2	6024	**00:53**
fabric.p4	V1Model	958	**00:02**
switch.p4 (from $P4_{14}$)	V1Model	5894	**10:20**

Table 4: Time needed to get semantics from a $P4_{16}$ program.

Hence, our Tofino bug count does not include any front- and mid-end crash and semantic bugs already present in P4C. We also do not include Tofino compiler crashes that were caused by a missed transformation in the P4C front end. The Tofino back end was relying on these passes to correctly transform specific P4 expressions. We filed two of these crashes in the Tofino compiler as missed optimization issues in P4C.

Fixing the bugs. Out of the 96 new bugs we filed, 54 have been fixed. The remaining bugs have been assigned a developer, but are still open because we filed them very recently, they required a specification change to be resolved first, or they have been de-prioritized in favor of more pressing bug reports. We have received confirmation by the Tofino compiler developers that 8 bugs have already been resolved; the remainder are targeted to be resolved by the next release.

7.2 Performance on Large P4 Programs

We also measured the time Gauntlet currently requires to generate semantics for several large P4 programs (Table 4). Generating semantics is the slowest part of our validation check; comparing the equality of the generated formulas in Z3 is typically fast. We have observed that retrieving semantics for a single pass takes on the order of a minute for a large program. We believe we can substantially improve this performance for two reasons. First, large parts of our semantic interpreter are written in Python as opposed to C++. Second, we currently use a simple state-merging approach for parser branches. This approach does not sufficiently address the scaling challenge of dense branching. When run on `switch.p4` retrieving semantics takes about 10 minutes. We note, however, that `switch.p4` is not a representative switch program as the code is autogenerated from old $P4_{14}$ code. Programs like `switch_tofino_x0.p4`, which model the data plane of a data center switch, only require a minute per pass.

7.3 Deep Dive into Bugs

Ripple effects. A common crash we observed occurs because a compiler pass incorrectly transforms an expression or does not process it at all. Back end compiler developers rely on the front end to correctly transform the IR of the P4 program. But, if a pass misses a language construct it is responsible for, the back end often cannot handle the resulting expression and

```
1 control ig(inout Hdr h, ...) {
2   apply {
3     h.mac_src =
4     (h.mac_src > 2 ? 48w1 : 48w2) + h.mac_src;
5   }
6 }
```

(a) A bug caused by a defective pass.

```
1 control ig(inout Hdr h, ...) {
2   apply {
3     h.mac_src = (1 << h.modifier) + 8w1;
4   }
5 }
```

(b) A crash in the type checker.

```
1 control ig(inout Hdr h, ...) {
2   apply {
3     bool tmp = 1 != 8w2[7:0];
4   }
5 }
```

(c) An incorrect type checking error.

```
1 control ig(inout Hdr h, ...) {
2   action assign_eth_type(inout bit<8> val) {
3     h.eth_type[15:8] = 0xFF;
4   }
5   apply {
6     assign_eth_type(h.eth_type[7:0]);
7   }
8 }
```

(d) Incorrect deletion of an assignment.

```
1 control ig(inout Hdr h, ...) {
2   apply {
3     h.ipv4.setInvalid();
4     h.ipv4.src_addr = 1;
5     h.eth.src_addr = h.ipv4.src_addr;
6     if (h.eth.src_addr != 1) {
7       h.ipv4.setValid();
8       h.ipv4.src_addr = 1;
9     }
10  }
11 }
```

(e) An unsafe compiler optimization.

```
1 control ig(inout Hdr h, ...) {
2   action assign_and_exit(inout bit<16> val) {
3       val = 0xFFFF;
4       exit;
5   }
6   apply {
7     assign_and_exit(h.eth_type);
8   }
9 }
```

(f) Incorrect interpretation of exit statements.

Figure 5: Examples of bugs that were caught by Gauntlet.

generates an assertion failure. For example, in program 5a, the front end `SideEffectOrdering` [10] pass should have converted the conditional operator in line 3 into normal if-then-else control flow. However because of the addition ex-

pression, the pass failed to transform the conditional operator, which ultimately caused an assertion to fire in the Tofino back end [61]. In another case, the `InlineFunctions` [10] pass did not fully inline all functions calls, causing a crash in back ends that were not able to understand function calls and expected them to have been inlined by then [58].

Crashes in the type checker. Many of the crashes (21 out of 33) were in the type checker infrastructure. The code in 5b shows an expression that crashed type checking [60]. It is not possible to shift this value since its width is unknown at compile-time. This program was deemed illegal, but the specification did not explicitly forbid it. The type checker tried to infer a type regardless and crashed. This bug also triggered an update to the $P4_{16}$ specification [27]. In other cases, the type checker was incorrectly forbidding a valid expression. In example 5c, the program was legal, but because a safety check in the `StrengthReduction` [10] pass was incorrectly implemented, the resulting slice index was overflowing and turned negative, which prompted the type checker to terminate with an error message [60].

Handling side effects. Side effects from a function operate on the concept of copy-in/copy-out semantics, described earlier. However, these semantics, while seemingly simple, turn out to be hard to implement correctly in the compiler. A particularly tricky case can be seen in 5d [64].

In the program, a slice of a variable is passed as an `inout` parameter. At the same time, a disjoint subset of the variable is assigned within the function. The correct behavior here is to leave the assignment unchanged, and copy back the sliced portion of the variable alone. However, the compiler assumed that the entire variable would be copied back and removed the assignment in line 3, an incorrect optimization.

A large subset of the semantic bugs we found in P4C (at least 11 out of 26) can be traced to incorrect handling of side effects and copy-in/copy-out. Copy-in/copy-out is difficult to handle because for a compiler pass that reorders expressions or statements, side-effects can be translated incorrectly.

Unstable code. Even though the $P4_{16}$ language has limited undefined behavior, we also found incidents of unstable code [73]. This unstable code conforms with the specification but may lead to instability in specific back end targets. Dumitru et al. also discuss the potential safety consequences of undefined variable access [24]. Program 5e is a concrete example. The compiler collapses the assignment of line 4 into line 5, setting `h.eth.src_addr`, which is still part of a valid header, to 1. All of this is legal behavior, since read and write operations on invalid header values are undefined as part of the P4 specification. The compiler is free to perform these optimizations. However, these changes may cause issues in specific back ends, e.g., back ends in which assignments to invalid headers are no-ops. In this case, the compiler has chosen a particular subset interpretation of undefined behavior, which may clash with the expectations of programmers for that back end. We raised this with the compiler developers, who agreed to print a warning [62].

Consequences of compiler changes. Once we started actively monitoring the master branch of P4C we observed that many (19 out 59) of the bugs we filed in P4C were caused by recent merges into master. A notable example is a recent change to the `Predication` [10] pass, which caused at least 6 (1 crash and 5 semantic) new bugs. We caught and filed these bugs quickly during our weekly routine random code generation. The compiler pass has become so complicated that the compiler maintainers are now relying on Gauntlet to ensure correctness [3]. A P4 programmer also filed a bug on this issue [28]. The report was considered a duplicate because of our earlier reports, highlighting that the bugs we find do affect actual P4 programmers.

Specification changes. Some of our bug reports kicked off larger discussions and changes around the P4 language specification. Our bug reports and questions have led to at least 6 distinct specification changes. For example, a concern we had about the validity of uninitialized headers (at what point does a header variable become valid?) led to three clarification pull requests on the specification and a suggestion to propose more fundamental changes for the next language version [30].

Another prominent example was caused by ambiguity in the specification. In example 5f, the `RemoveActionParameters` [10] compiler pass moved the statement in line 3 after the exit statement, because the assumption was that exits called within functions ignores the copy-in/copy-out semantics. We instead interpreted exit statements to still respect copy-in/copy-out semantics and caught the discrepancy. This is a significant difference. A packet that traverses the control program could lose all the modifications that have been written to its header, a potential security risk. We filed this as a concern with the open-source community [59] and our interpretation was deemed reasonable, which required a specification update [31]. The corresponding compiler changes resulted in at least 3 new bugs, which we detected and filed.

Invalid transformations. Because P4C provides the option to emit transformed programs after each pass as a valid P4 program, the compiler developers maintain an invariant that each compiler pass in the front and mid end needs to emit syntactically correct P4. We uncovered several bugs with how P4 code is emitted and transformed across compiler passes. We detected these bugs by reparsing each P4 program after it had been emitted by the `ToP4` compiler module. If the emitted program can not be reparsed, it indicates a bug in one of three compiler components: the `ToP4` module, the P4C parser, or the compiler pass. While these bugs typically do not harm correctness, they affect compiler debugging. Overall, we identified 4 bugs of invalid intermediate P4, all of which were fixed; these 4 are not included in our count of 96. Additionally, because we reparse P4 after each compiler pass, we found a

case where the emitted program being parsed incorrectly was a symptom of a larger bug in the P4C parser [63].

7.4 Lessons Learned

P4C debugging support. P4C has several facilities that were useful for bug finding. The ability to dump the intermediate representation, specify which passes to dump, and the `ToP4` tool, which converts the P4 IR to P4 programs accelerated our development process. In addition, the compiler has comprehensive assert instrumentation with distinct messages, which we used to identify unique crash bugs and to distinguish them from valid error messages. The AST visitor library in P4C allowed us to develop extensions like our random program generator and interpreter.

P4C's nanopass architecture, which factors the compiler into a large number of "thin" passes, helps with bug fixing, especially for semantic bugs that were narrowed down to one pass by translation validation. A different architecture that has fewer "thick" passes would need more developer effort to fix semantic bugs. We also observed that almost all crash bugs were assertion violations where an invariant was violated in a particular compiler pass due to an incorrect or absent compiler transformation from a previous pass. In the absence of such assertions, these crash bugs could have easily manifested as semantic bugs that are harder to detect.

Reporting bugs. This project would not have been possible without the responsiveness and receptiveness of the P4 community. Our questions, concerns, and bug reports were answered within a day and in great detail. The developers were able to even dissect our initial questions and confusions into bug reports, guiding us further in our development effort. We were encouraged to participate in the language design working group that discusses changes to the P4 specification.

Likewise, when we filed bugs for the closed-source and proprietary Tofino compiler, we found the developers to be receptive and responsive. Still, the pace of bug finding and fixing with the Tofino compiler was slower than the open-source compiler because of two unavoidable reasons. First, we naturally didn't have access to the company bug tracker to assess the life cycle of our bug once it had been filed. Second, the official binary of the Tofino compiler updates less frequently than P4C, which can be rebuilt from source after every commit. Hence, we would trigger the same bugs repeatedly in our testing until a new Tofino compiler version with a bug fix was released. Neither of these two problems would manifest, if our tool was to be used internally as part of the compiler development process for Tofino.

8 Future Work

New types of bugs. Gauntlet can not find compiler bugs that affect performance or resource usage of generated code. For a switching ASIC that guarantees line-rate performance, the compiler must produce code that consumes a small number of computational and memory units [33]. For software targets where line rate performance is not guaranteed, the generated code must have good performance. For example, the P4-eBPF compiler, which converts P4 to eBPF/XDP [35] byte code, occasionally produces code with poor performance [72]. We are investigating methods that allow us to identify when a compiler pass negatively affects performance and resource usage. We anticipate that handling such bugs would require techniques that are conceptually very different from our methods, which deal with correctness bugs.

Supporting aggressive compiler optimizations. Similar to credible compilation [55], we plan to repurpose Gauntlet as an attachable compiler plugin to facilitate development of experimental compiler optimizations. During compilation, if a newly added optimization produces semantically incorrect code, Gauntlet will notify the compiler to discard the optimization. With this technique, a developer can integrate potentially buggy code into the compiler while still guaranteeing a safe compilation process. However, for the plugin to be useful, Gauntlet's translation validation needs to be fast enough so that compilation time remains acceptable.

Extending translation validation to the compiler back end. So far we have applied translation validation only to compiler front and mid ends. This is because these passes allow us to dump the P4 program before and after the pass has run, allowing us to compare the before and after programs for equality. The back end is typically proprietary, inaccessible, and uses an opaque intermediate representation. To understand the constraints of these back ends we are currently working with industry compiler developers to integrate translation validation into their compilers. We will develop translation validation techniques that allow us to compare a P4 program's semantics with the semantics of a back end language that is not P4.

Long-term study on translation validation in CI. Now that translation validation is running as part of the CI pipeline of P4C we would like to perform empirical, long-term studies. We want to identify which passes frequently cause semantic issues and understand why they do. We would also like to observe how developer-friendly our tool is. For example to avoid confusing compiler developers, we already had to make sure that Gauntlet does not report changes in undefined behavior [29] or fails gracefully when Gauntlet does not support a particular language construct [11].

Automatic test case reduction. We have not developed an automatic test-case reduction suite (e.g., C-Reduce [54]) and reduce buggy programs in a manual fashion. After our testing pipeline has identified problematic programs in a randomly generated batch, we inspect each P4 program individually. We prune the random P4 program that caused the bug until we get a sufficiently small program that can be attached to a

bug report. We are currently automating this process.

Better coverage of the compiler and P4$_{16}$ language. While our symbolic interpreter provides semantics for the majority of the P4$_{16}$ language constructs, we currently do not generate programs that contain several P4$_{16}$ language features: extern calls, method overloading, type definitions, variable bit vectors, run-time indices, match types such as longest prefix or ternary matches, type-inference for generic types in function bodies, annotations, and various custom table properties. We expect that adding most of these will be conceptually straightforward, although adding each language construct is a fair amount of additional engineering. One particular construct that we anticipate being hard to support is externs. While our interpreter includes an extension model to add custom semantics for each extern, extern behavior is very back-end-specific. It is hard to develop accurate semantics for these externs without detailed hardware knowledge of each target. We also do not track how much of the compiler source code we actually cover with our program generator. For future work, we would like to measure the compiler code coverage of a generated P4 program with `gcov` to understand avenues for improvement.

9 Related Work

P4K [38] was an effort to formalize the P4 language using the K-framework [56]. In the process of defining these semantics, the authors found several issues in the P4 specification. P4K supports the use of translation validation similar to our tool. `netdiff` [23] uses symbolic execution to verify the equivalence of data planes, such as those written in P4. They do so by converting P4 and other data plane programs into the SEFL language [69], which in turn can be converted to Z3. The Z3 expressions corresponding to different data planes can then be compared for equality. `netdiff`'s equivalence checking technique is comparable to our translation validation technique. However, neither P4K nor `netdiff` were explicitly designed for finding compiler bugs. To enable such bug finding, we need both a source of random P4 programs and a translation validation technique to compare intermediate versions of these programs. Further, for some back ends such as the Tofino compiler, translation validation is insufficient, requiring us to use model-based testing instead.

`p4pktgen` [49] is a P4 test-case generation tool, similar to our model-based testing technique. `p4pktgen` parses the JSON file generated by the BMv2 back end and outputs a Z3 formula, which it uses to create test cases. Using `p4pktgen`, the authors were able to find several bugs in how BMv2 executes JSON files. However, because it operates on output JSON instead of the input P4 program, unlike Gauntlet, `p4pktgen` can not find bugs in intermediate compiler passes.

petr4 [21] is a project with the goal of providing independent and complete formal foundations for the P4$_{16}$ language. petr4 is complementary to our work. While we are explicitly targeting the official P4$_{16}$ compiler and specialized our tools to find bugs during compilation, petr4 aims to find inconsistencies and mistakes in the official P4$_{16}$ specification and type system. petr4 provides an interpreter that aims to establish unambiguous semantics for a given P4$_{16}$ program. This semantic interpretation can potentially be used to guide the development of our own interpreter semantics.

10 Conclusion

This paper presented Gauntlet, a tool for finding bugs in packet-processing compilers for languages such as P4. Gauntlet combines random program generation, translation validation, and model-based testing to find both crash and semantic bugs in P4 compilers. It has been highly effective, uncovering 96 new and confirmed bugs. 54 of these have been fixed and the rest have been assigned to a compiler developer. We have open sourced Gauntlet at `p4gauntlet.github.io` and it now runs as part of the CI infrastructure of P4C.

While we developed Gauntlet for P4, we believe the core technique that makes Gauntlet effective is much more general. In particular, Gauntlet exploits the fact that P4 is a DSL with significant restrictions such as the lack of loops. These restrictions allow us to revive and simplify prior techniques such as translation validation and take them much further in the context of a DSL. For example, to our knowledge, Gauntlet is the first instance of translation validation running as part of a compiler's CI infrastructure. We believe this ability to exploit domain specificity for more effective compiler bug finding will increasingly be applicable to other DSLs beyond P4.

Acknowledgements

We would like to thank our shepherd, Madan Musuvathi, and the anonymous OSDI reviewers for their valuable feedback. We would also like to thank Amy Ousterhout, Aurojit Panda, Thomas Wies, Michael Walfish, Srinivas Narayana, and Mihai Budiu for their insightful feedback on paper drafts and the project. We are grateful to the P4 compiler team at Barefoot Networks and the open-source P4 community for their feedback and willingness to engage with our bug reports. In particular we would like to thank Mihai Budiu, Nate Foster, Andy Fingerhut, Han Wang, and Antonin Bas for their prompt responses to our many bug reports. We also thank Aatish Varma and Peixuan Gao for experimenting with using AFL for finding bugs in P4C as part of their course project.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al.

Tensorflow: A system for large-scale machine learning. In *USENIX OSDI*, 2016.

[2] Andrei Alexandru Agape, Mădălin Claudiu Dănceanu, René Rydhof Hansen, and Schmid Stefan. P4Fuzz: Compiler fuzzer for dependable programmable dataplanes. In *ACM ICDCN*, 2021.

[3] anasyrmia. Fix: Predication issue #2345. https://github.com/p4lang/p4c/pull/2564, 2020. Accessed: 2020-10-15.

[4] Barefoot. Industry-first co-packaged optics Ethernet switch. https://www.barefootnetworks.com/technology/. Accessed: 2020-10-15.

[5] Antonin Bas. PTF: Packet testing framework. https://github.com/p4lang/ptf. Accessed: 2020-10-15.

[6] Antonin Bas. The reference P4 software switch. https://github.com/p4lang/behavioral-model. Accessed: 2020-10-15.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[8] Broadcom. Trident4 / BCM56880 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series. Accessed: 2020-10-15.

[9] Broadcom. NPL: Open, high-level language for developing feature-rich solutions for programmable networking platforms. https://nplang.org/, 2019. Accessed: 2020-10-15.

[10] Mihai Budiu. The $P4_{16}$ reference compiler implementation architecture. https://github.com/p4lang/p4c/blob/master/docs/compiler-design.pptx, 2018. Accessed: 2020-10-15.

[11] Mihai Budiu. Tuple elim. https://github.com/p4lang/p4c/pull/2451, 2020. Accessed: 2020-10-15.

[12] Mihai Budiu and Chris Dodd. The $P4_{16}$ programming language. *ACM SIGOPS Operating Systems Review*, 2017.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX OSDI*, 2008.

[14] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM Computer Communication Review*, 2007.

[15] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *ACM/IEEE ICSE*, 2016.

[16] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: A new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*, 1998.

[17] Cisco. Cisco Silicon One. https://www.cisco.com/c/en/us/solutions/service-provider/innovation/silicon-one.html. Accessed: 2020-10-15.

[18] The P4.org consortium. *The $P4_{16}$ Language Specification, version 1.2.1*, June 2020.

[19] Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. Model-based testing in practice. In *ACM/IEEE ICSE*, 1999.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[21] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexandar Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: Formal foundations for P4 data planes. In *ACM POPL*, 2021.

[22] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: Towards bug-free P4 programs. In *ACM SIGCOMM*, 2020.

[23] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *USENIX NSDI*, 2019.

[24] Mihai Valentin Dumitru, Dragos Dumitrescu, and Costin Raiciu. Can we exploit buggy P4 programs? In *ACM SOSR*, 2020.

[25] Matthias Eichholtz, Eric Campbell, Nate Foster, Guido Salvaneschi, and Mira Mezini. How to avoid making a billion-dollar mistake: Type-safe data plane programming with SafeP4. *arXiv preprint arXiv:1906.07223*, 2019.

[26] Andy Fingerhut. Behavioral model targets. https://github.com/p4lang/behavioral-model/blob/master/targets/README.md, 2018. Accessed: 2020-10-15.

[27] Andy Fingerhut. Forbid shifts with unknown widths. https://github.com/p4lang/p4-spec/pull/814, 2020. Accessed: 2020-10-15.

[28] Andy Fingerhut. Incorrect transformation in predication pass. https://github.com/p4lang/p4c/issues/2345, 2020. Accessed: 2020-10-15.

[29] Andy Fingerhut. Make stricter PSA tests that verify packet_path and instance fields. https://github.com/p4lang/p4c/pull/2509, 2020. Accessed: 2020-10-15.

[30] Andy Fingerhut. Reducing requirements for initializing headers. https://github.com/p4lang/p4-spec/issues/849, 2020. Accessed: 2020-10-15.

[31] Andy Fingerhut. Specify that copy-out behavior still occurs after return/exit statements. https://github.com/p4lang/p4-spec/pull/823, 2020. Accessed: 2020-10-15.

[32] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in P4 programs with assertion-based verification. In *ACM SOSR*, 2018.

[33] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, 2020.

[34] Chris Hawblitzel, Shuvendu K Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In *ACM ESEC/FSE*, 2013.

[35] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress Data Path: Fast programmable packet processing in the operating system kernel. In *ACM CoNEXT*, 2018.

[36] Jean D Ichbiah, Bernd Krieg-Brueckner, Brian A Wichmann, John GP Barnes, Olivier Roubine, and Jean-Claude Heliard. Rationale for the design of the Ada programming language. *ACM SIGPLAN notices*, 1979.

[37] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, et al. Crellvm: Verified credible compilation for LLVM. In *ACM PLDI*, 2018.

[38] Ali Kheradmand and Grigore Rosu. P4K: A formal semantics of P4 and applications. *arXiv preprint arXiv:1804.01468*, 2018.

[39] Ariel Kit. Programming the entire data center infrastructure with the NVIDIA DOCA SDK. https://developer.nvidia.com/blog/programming-the-entire-data-center-infrastructure-with-the-nvidia-doca-sdk/. Accessed: 2020-10-15.

[40] Suriya Kodeswaran, Mina Tahmasbi Arashloo, Praveen Tammana, and Jennifer Rexford. Tracking P4 program execution in the data plane. In *ACM SOSR*, 2020.

[41] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 2014.

[42] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *ACM OOPSLA*, 2015.

[43] Xavier Leroy. Formal certification of a compiler backend or: Programming a compiler with a proof assistant. In *ACM POPL*, 2006.

[44] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. p4v: Practical verification for programmable data planes. In *ACM SIGCOMM*, 2018.

[45] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *ACM PLDI*, 2015.

[46] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 1998.

[47] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[48] George C Necula. Translation validation for an optimizing compiler. In *ACM PLDI*, 2000.

[49] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. p4pktgen: Automated test case generation for P4 programs. In *ACM SOSR*, 2018.

[50] Brian O'Connor, Yi Tseng, Maximilian Pudelko, Carmelo Cascone, Abhilash Endurthi, You Wang, Alireza Ghaffarkhah, Devjit Gopalpur, Tom Everman, Tomek Madejski, et al. Using P4 on fixed-pipeline and programmable Stratum switches. In *ACM/IEEE ANCS*, 2019.

[51] Pensando. A new way of thinking about next-gen cloud architectures. https://p4.org/p4/pensando-joins-p4.html. Accessed: 2020-10-15.

[52] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998.

[53] GNU Project. gcov–a test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 1987. Accessed: 2020-10-15.

[54] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *ACM PLDI*, 2012.

[55] Martin C Rinard. Credible compilation. Technical report, Massachusetts Institute of Technology, 2003.

[56] Grigore Roșu and Traian Florin Șerbănuță. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 2010.

[57] Fabian Ruffy. Add Travis validation tests for P4C. https://github.com/p4lang/p4c/pull/2458. Accessed: 2020-10-15.

[58] Fabian Ruffy. BMV2 backend compiler bug unhandled case. https://github.com/p4lang/p4c/issues/2291, 2020. Accessed: 2020-10-15.

[59] Fabian Ruffy. Calling exit in actions after an assignment. https://github.com/p4lang/p4c/issues/2225, 2020. Accessed: 2020-10-15.

[60] Fabian Ruffy. Compiler bug: Null cst. https://github.com/p4lang/p4c/issues/2206, 2020. Accessed: 2020-10-15.

[61] Fabian Ruffy. Missing StrengthReduction for complex expressions in actions. https://github.com/p4lang/p4c/issues/2279, 2020. Accessed: 2020-10-15.

[62] Fabian Ruffy. More questions on setInvalid. https://github.com/p4lang/p4c/issues/2323, 2020. Accessed: 2020-10-15.

[63] Fabian Ruffy. Question about parser behavior with right shifts. https://github.com/p4lang/p4c/issues/2156, 2020. Accessed: 2020-10-15.

[64] Fabian Ruffy. SimplifyDefUse incorrectly removes assignment in actions with slices as arguments. https://github.com/p4lang/p4c/issues/2147, 2020. Accessed: 2020-10-15.

[65] Dipanwita Sarkar, Oscar Waddell, and R Kent Dybvig. A nanopass infrastructure for compiler education. *ACM SIGPLAN Notices*, 2004.

[66] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *ACM OOPSLA*, 2013.

[67] Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Stefan Schmid, Artur Hecker, and Anja Feldmann. Towards runtime verification of programmable switches. *arXiv preprint arXiv:2004.10887*, 2020.

[68] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with Vera. In *ACM SIGCOMM*, 2018.

[69] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM*, 2016.

[70] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *ACM POPL*, 2009.

[71] The XLA Team. XLA – TensorFlow compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html, 2017. Accessed: 2020-10-15.

[72] William Tu, Fabian Ruffy, and Mihai Budiu. P4C-XDP: Programming the linux kernel forwarding plane using P4. In *Linux Plumbers Conference*, 2018.

[73] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *ACM SOSP*, 2013.

[74] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM PLDI*, 2011.

[75] Michał Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2020-10-15.

[76] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A translation validator for optimizing compilers. *Electronic notes in theoretical computer science*, 2002.





Aragog: Scalable Runtime Verification of Shardable Networked Systems

Nofel Yaseen[◇], Behnaz Arzani[†], Ryan Beckett[†], Selim Ciraci[§], and Vincent Liu[◇]
[◇]*University of Pennsylvania* [†]*Microsoft Research* [§]*Microsoft*

Abstract

Network functions like firewalls, proxies, and NATs are instances of distributed systems that lie on the critical path for a substantial fraction of today's cloud applications. Unfortunately, validating these systems remains difficult due to their complex stateful, timed, and distributed behaviors.

In this paper, we present the design and implementation of *Aragog*, a runtime verification system for distributed network functions that achieves high expressiveness, fidelity, and scalability. Given a property of interest, *Aragog* efficiently checks running systems for violations of the property with a scale-out architecture consisting of a collection of global verifiers and local monitors. To improve performance and reduce communication overhead, *Aragog* includes an array of optimizations that leverage properties of networked systems to suppress provably unnecessary system events and to shard verification over every available local and global component. We evaluate *Aragog* over several network functions including a NAT Gateway that powers Azure, identifying both design and implementation bugs in the process.

1 Introduction

An emerging bottleneck to correctness and availability in modern cloud systems are the various network functions (*e.g.*, firewalls, NATs, and load balancers) that interpose on the majority of application requests flowing to, from, and between servers in the cloud. Over time, these network functions (NFs) have become increasingly complex. Today, many of these functions are full-fledged distributed systems whose correctness depends on the coordination of multiple devices as well as on stored state and system timing.

Configuration errors and software bugs in these components can have an outsized impact on SLAs [4] not only because of the complexity of these systems, but also because they are on the critical path of most application requests. For instance, a production NAT gateway we verify in this work manages (replicated) states for millions of flows and errors in this system can lead to black holes, broken connectivity, forwarding loops, and more. Public incident reports from providers show multiple outages due to errors like these [4, 19].

To improve availability, recent proposals suggest using *static verification* to prove the correctness of these systems [21, 25, 29, 34, 40–42, 44]. While powerful, the need to reason about every possible interleaving of inputs and control flows presents a significant obstacle to the application of these techniques in today's network functions. Attempting to explore the full space of control flow paths often leads to state/path explosion [25, 29, 40]. Mitigations to this problem, broadly speaking, can be categorized in a few ways. The first is to require the use of special programming languages or other types of programmer interaction [21, 43]. The second is to use model checking techniques to more efficiently explore all possible system behaviors. Finally, many systems—to reduce the state space they must verify and to make verification more tractable—limit the set of verifiable behaviors, *e.g.*, to those that are unordered [34], abstract [10], or restricted to a single machine [42, 44].

While effective in many cases, each of these approaches also comes with significant drawbacks. With the first, programmers are saddled with a substantial burden that can overwhelm the development of the system. With the second, model checking still typically relies on hand-written models of functionality, which may be difficult to provide for a rapidly evolving or complex system. Finally, limiting the scope of verification fails to extend to the increasingly complex services found in modern networks—services that arguably need verification the most.

An alternative approach to static verification is runtime verification of distributed systems. In runtime verification, a tool extracts information about the current state of a running system (testbed, canary, or production) to verify that invariants hold throughout execution [13, 14, 28, 30, 31, 33, 36, 39]. Compared to static verification, runtime verifiers only test inputs and control flows that are seen in practice, thus improving scalability and enabling verification of actual deployments running over actual data. In return, they sacrifice a principled exploration of the system's behavior and the ability to catch bugs early. We argue that these tradeoffs are a better fit for our operators' requirements.

We find today's runtime verifiers cannot be applied as-is to deployed network functions. The challenge (for network functions) is the need, at runtime, to: (1) reason about the coordination between events issued at different locations, (2) efficiently aggregate global state after each event, and (3) scale sub-linearly with the size of the original system—after all, a verifier that requires the same amount of resources as the system itself is untenable for most production environments.

In this paper, we present the design of a scale-out, runtime

verification tool for network functions called *Aragog* that overcomes the above challenges. *Aragog* provides a simple, but expressive language for describing violations of invariants, with a focus on supporting network functions. Examples of network-centric language features that are found in *Aragog*'s Invariant Violation (IV) specifications, but that are uncommon in other runtime verifiers are support for properties that are parametric over the "location" of events, properties that reference stateful variables, the ability to execute partial matches over packet fields, and support for temporal predicates.

Aragog translates these IV specifications to a set of symbolic automata that can efficiently verify the current global state of the system. In addition, to ensure that the system can scale out to a near-unlimited number of machines, *Aragog* implements the core of these checks on top of production stream processing systems [2, 3]. To efficiently coordinate between distributed verifiers, *Aragog* relies on hardware-supported time synchronization protocols like PTP. Finally, to minimize the overhead of the verification system, *Aragog* leverages observations that network events/invariants are typically:

Flow- or connection-based: For most network functions, correctness is defined on a per-flow or per-connection basis. From the IV specification, *Aragog* derives sharding keys that allow it to distribute the verification task across independent workers. These shards also expose boundaries on which we can gracefully scale down to a sampled subset of the input.

Partially suppressible: Rather than aggregate all events in the system to a logically centralized verifier, most network events have limited windows of relevance depending on the state of the system, e.g., only if the connection has recently been closed. *Aragog* includes an optimization scheme to suppress such messages before they ever leave the NF instance.

Aragog does not guarantee perfect accuracy under asynchrony—to do so would require atomicity guarantees in the critical path of the network functions. *Aragog* instead handles these situations speculatively and notifies users after-the-fact[1] about transient inconsistency (§7.3). Despite this, *Aragog* identified at least four bugs in an early (limited) deployment of a real distributed network function: Azure's new NAT gateway (NATGW). These bugs were detected within ∼100 ms of occurrence. Compare this to the hours our operators typically spend searching for similar bugs.

To summarize, our work makes the following contributions:

- We present a case study of the needs of a large modern network function from Microsoft's Azure. The system exhibits several interesting characteristics and suggests key requirements for verifier design.
- We synthesize ideas from timed regular expressions, symbolic automata, and parametric verification. To the best of our knowledge, ours is the first to demonstrate a concrete need and method for combining these concepts.
- We introduce the design and implementation of *Aragog*, a system for at-scale runtime verification. When needed, *Aragog* can also run on traces (offline) and therefore complement static verification to find implementation bugs in distributed networked systems. Among other innovations, *Aragog* includes a novel method for computing location-dependent suppression of network events.
- We introduce a collection of *Aragog* invariant violations for a set of distributed network functions, and we evaluate *Aragog* on NATGW and a distributed firewall.

[1] This reporting happens in under 1 s. This delay is on the same order as other alerting systems used in our production networks.

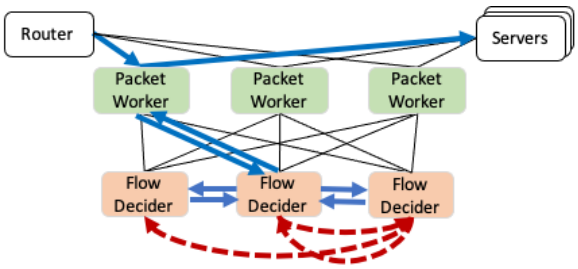


Figure 1: The architecture of our NATGW. The bolded blue arrows show the sequence of communication to handle the SYN packet of an incoming flow: it is sent to a random packet worker, which forwards it to the flow decider in charge of that flow. The flow decider chooses a target server and replicates the mapping to other deciders, then installs it in the original packet worker. The three dashed red arrows trace the allocation of the mapping for the reverse flow.

2 Motivation: A Cloud-scale NAT Gateway

Our work is grounded in experience with Azure's large-scale NF that we call NATGW. NATGW is a cloud-scale NAT gateway that balances incoming requests over available servers and supports almost all external traffic.

Like many other NFs of similar scale [16, 35], NATGW is implemented entirely in software, is distributed across a pool of servers, and replicates state for fault tolerance. Routers use ECMP-based anycast to randomly direct packets to NATGW workers, which then rewrite the destination IP and port to point at a target server. A similar translation occurs for packets in the reverse direction (from the server to the client).

Figure 1 depicts the NATGW architecture. It is composed of two types of nodes: packet workers and flow deciders. Packet workers process every packet passing through the NATGW, parsing its header, looking up the target server, and rewriting the packet header to point to that target. The mapping of a flow to a target server is decided with the help of a sharded set of flow deciders. The deciders cache and replicate these mappings to other deciders to ensure availability.

Flow allocation. When a packet worker receives the first packet of a new flow, it uses a hash of the 5-tuple to identify

the "primary" flow decider that owns the flow and forwards the packet to that decider. The primary then:

1. Decides the target server to which to send the new flow and installs the mapping in the local flow cache.
2. Sends the reverse mapping to the flow decider that "owns" the other end of the flow. Together, these two mappings cover translation for both incoming and outgoing traffic.
3. With its counterpart primary, greedily copies the mappings to the cache of other flow deciders in a manner akin to chain replication: decider i will try to copy to deciders $(i+1) \bmod N$ and $(i+2) \bmod N$, where N is the number of deciders. If one is down, it switches to $(i+3) \bmod N$.
4. Installs the mapping into the originating packet worker.

After the above flow allocation, the packet worker can process all subsequent packets of the flow without coordination with any other node. If the packet worker fails, anycast redirects the packet to another worker; the new worker will send the packet to the primary flow decider, fetching the existing mapping. If the flow decider fails, packet workers will query the next deciders in the sequence until they find the mapping.

Flow mapping timeouts. All components time out their flow mappings to ensure stale entries are eventually removed.

To ensure NATGW maintains mappings for active flows, packet workers periodically send a keepalive message to the primary decider. The primary forwards the keepalive to all replicas, refreshing the timeout on every instance of the mapping in the system. In parallel, the primary forwards the keepalive to the primary in charge of the reverse mapping.

Eventual consistency. This NATGW design exhibits some interesting properties. One of them is a choice to allow for temporary inconsistency in the presence of node failures in order to satisfy certain practical and performance constraints.

For example, consider three replicas of a flow mapping R_P, R_{P+1}, and R_{P+2}, where R_P is the primary. To delete the mapping, R_P would send a delete request to both of the other nodes. Now imagine the message to R_{P+1} is dropped. Rather than waiting for R_{P+1}, the others will go ahead and delete f. If, later, R_P fails, packet workers will contact R_{P+1} for the mapping, which will return a stale/inconsistent result until a timeout or periodic sync eliminates the inconsistency.

There are known mitigations to the above behavior (*e.g.*, querying a quorum on every packet or initiating a view change algorithm on R_P's failure); however, these come with significant performance costs. Instead, the NATGW is an example of a *deployed* architecture that chooses eventual consistency after careful consideration of its drawbacks and alternative solutions. Our work is motivated by our operators' experience with such behaviors.

3 Design Goals

Our runtime verifier targets the following design goals:

Practicality. Network functions are complex; written in a variety of languages; and frequently rely on external libraries, drivers, and other components. NATGW, for example, is built using libraries like DPDK and interacts with an ecosystem of networking hardware and configurations. The intricacies of the systems, the richness of their dependencies, and the rapid evolution of all the associated components mean the system is not easily modeled or accurately simplified. Instead, verification should be of the end-to-end system, *in situ*.

In the same vein, *Aragog* should not place undue burden on developers, *e.g.*, by requiring engineers to perform non-trivial proof writing (as mandated by many deductive reasoning techniques). NATGW has over 40 thousand lines of code—*Aragog* should avoid incurring a proportional overhead.

Expressiveness. Prior work has observed a gap between state-of-the-art verification tools and the requirements of modern networks [33]. In particular, it is challenging to specify invariants related to: (1) parametric variables over values like locations or identifiers, (2) coordination between network devices, and (3) timing of events. Moreover, since the number of devices (*e.g.*, flow deciders) may vary over time as the system scales out, it is useful to express properties in a way that does not require explicitly naming components. *Aragog* should provide syntax and semantic support for these behaviors.

Scalability. Just as a single machine cannot handle all traffic entering a large network, it also cannot be expected to verify the correctness of the entire network. Rather, the verifier should scale out to arbitrary size and require fewer resources than the original system. Therefore, *Aragog* should attempt to minimize the number of messages exported from each NF, *e.g.*, by exporting events (resulting from the execution of the NF) rather than packets (the inputs to the NF).

Graceful degradation of accuracy. As we describe in Section 7.3, perfect precision and recall is impossible in an asynchronous system without substantial overhead. Instead, *Aragog*'s correctness goal is in the same spirit as NATGW's: perfect recall under the assumption of 'partial synchrony' [15] and notifications of potential false positives/negatives after-the-fact. Our operators find this is sufficient for most cases.

Near-real-time alerts. Diagnosing bugs manually can take hours of operator time and the network could worsen the longer the bug persists: *Aragog* should raise alerts within seconds of observing the offending sequences of events.

4 *Aragog*'s Architecture

We present the design and implementation of a practical, expressive, and scalable verifier for large and complex NF deployments. Our system, *Aragog*, is a combination of a language for specifying invariant violations and a scale-out runtime system. *Aragog* takes a grey-box approach, requiring small changes to the underlying source code in order to export events of interest to the verifier. Thus, *Aragog* verifies by:

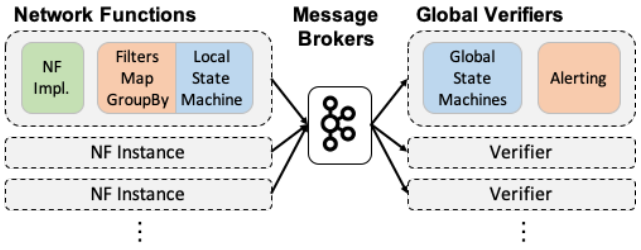


Figure 2: The architecture of *Aragog*. NF instances generate and feed events into a set of local state machines. The NF instances use these state machines to determine if they can hide unnecessary messages before exporting the rest to the global verifier. These messages pass through a Kafka cluster and are streamed to a set of Flink-based verification engines.

Specifying invariant violations over user-defined events. To provide operators with sufficient expressiveness to check network-level events, *Aragog* comes equipped with a new language for specifying invariant violations that is based on writing symbolic regular expressions over a global trace of events (and their locations) in the system. *Aragog*'s language includes a notion of parameterized "variables" that allows violations to be described in a way that holds for any combination of variable instantiations subject to constraints.

Checking for invariant violations. NF developers export any relevant events to *Aragog*. To scale up checking of the event stream, *Aragog* does two things. The first is to automatically analyze and split verification into local and global components. The local level resides at the NF instances themselves, where *Aragog* infers (only using the state of the local instance) whether it can safely suppress the event before exporting it to the global *Aragog* verifier. The second is to leverage the fact that most network invariants are defined across *related flows* rather than globally—for instance, on the granularity of a 5-tuple. As a result, events can be automatically sharded across a cluster of scale-out stream processing workers using Kafka [26] and Flink [11].

Note that, because the invariants are defined and checked only across related flows, we only need to know the correct ordering for events pertaining to those flows: event timestamps that use the sub-microsecond-scale synchronization of PTP suffices for our needs. For many production networks, these types of event exports are already common.

Overview. Figure 2 shows *Aragog*'s design. Users describe a set of invariant violations that identify classes of incorrect behavior. *Aragog* translates these to a set of symbolic automata and then splits the automata into local and global components. It then deploys these to NF instances and global verifiers.

At runtime, NF instances stream events into the pipeline. The local *Aragog* agent filters, maps, and shards events The message brokers aggregate and compact those streams The global verifiers determine, for the shard, whether a violation occurred. Kafka and Flink will automatically allocate resources and load balance requests to ensure scalability.

```
1  { "fields" : [
2      {"eventType" : 16},
3      {"nodeType" : 8},
4      {"sourceIPv4or6" : 8},
5      {"sourceIPv4or6==4" : [ {"srcIP" : 32} ],
6       "sourceIPv4or6==6" : [ {"srcIP" : 128} ]},
7      ...
8  ],
9  "constants" : {
10     "NAT_ALLOCATION" : 1,  // eventTypes
11     "FLOWCACHE_CONSENSUS" : 769,
12     "PACKET_WORKER" : 0,   // nodeTypes
13     ...
14 }}
```

Figure 3: A snippet of the NATGW JSON event schema.

5 Specification Language

Users define both events and policies over the events using two types of specifications that are inputs to *Aragog*: event definitions and Invariant Violation (IV) specifications. While both of these require the user to have some knowledge of the inner workings of the NF to specify how it can fail, our network operators determined that event-based violations struck a reasonable balance between precision and ease-of-use.

5.1 Event Definitions

Users specify the format of the event messages that arrive at the local verifier. *Aragog* expects these messages to be in the form of packed arrays of raw binary data whose format is defined with a JSON configuration file. For example, Figure 3 shows a selected subset of the definition for NATGW event messages. 'fields' contains the ordered list of expected fields in the message. Each field is defined by a JSON dictionary specifying the field's name and its length in bits—for instance, the first 16 bits of the event message is an `eventType`.

Conditionals. In addition to specifying the length of each field and their ordering, *Aragog* allows users to implement simple conditional parsing logic. The example event definition shows one such use where *srcIP* can be either IPv4 or IPv6. In the configuration shown, event messages include a 8-bit field that specifies the IP version number. Depending on the value of that version number, the next field is either a 32-bit or 128-bit `srcIP` field. These branches can define entire sub-headers and can contain nested conditionals.

Named constants. *Aragog* also allows users to define named constants representing integer values represented in decimal, hexadecimal, or binary notation. We show four such constants in Figure 3: two for values of the `eventType` field and one for the `nodeType` field. These are intended for use in IV specifications to make them more readable.

```
1 FILTER((eventType == FLOWCACHE_PRIMARY_ADD
2       || eventType == FLOWCACHE_REMOVE_ENTRY)
3       && workerType == FD)
4 GROUPBY(srcIP, dstIP, srcPort, dstPort, proto)
5 MATCH
6 (eventType == FLOWCACHE_PRIMARY_ADD) @ $X
7 ((eventType == FLOWCACHE_REMOVE_ENTRY) @ NOT $X)*
8 (eventType == FLOWCACHE_PRIMARY_ADD) @ NOT $X
```

Figure 4: An example IV specification that ensures at most one primary is ever active for a given flow.

5.2 Invariant-Violation (IV) Specifications

Aragog parses incoming event messages and checks them against a set of user-defined policies that describe sequences of events that violate the invariants of the system. Operators specify these policies using *Aragog*'s domain-specific language, which we detail in this subsection.

Figure 4 shows an example specification for our NATGW. The policy only pertains to a subset of events (lines 1–3), and *Aragog* verifies it on a per-5-tuple basis (line 4). A violation occurs when some node `$X` adds a primary mapping (line 6) and then a different node (`NOT $X`) adds the same mapping (line 8) without `$X` removing it. The full grammar for IV specifications is shown in Figure 5. Briefly, an IV specification consists of (1) a collection of event transformations followed by (2) a regex-like expression over the generated events.

5.2.1 Transformations

Aragog allows users to define a set of policy-specific transformations. In addition to enabling greater flexibility and expressiveness, *Aragog* also uses these transformations to perform an initial filtering and aggregation as well as to identify valid sharding strategies. *Aragog* currently supports three transformations: **GROUPBY**, **FILTER**, and **MAP**.

Operators can use **GROUPBY** to indicate which events need to be considered together and which can be considered separately. For example, when an operator wishes to guarantee at most one primary is active (Figure 4) for *each flow*, the **GROUPBY** is used to classify events into unique flows. *Aragog* uses this transformation to both simplify policy logic and to assist in the sharding of verification.

Operators can also use the **FILTER** transformation to indicate which events should be considered at all and which should be ignored. In the above example, we only care about flow deciders—specifically when they add a flow as a primary and when they delete the flow mapping from the cache; we can filter events of any other type or from any other type of node. **FILTER**s are critical for reducing the number of events handled by the verification framework.

Finally, operators can use the **MAP** transformation to generate entirely new fields based on mathematical expressions over existing fields of the event message.

⟨*IVspec*⟩ ::= ⟨*transformations*⟩ 'MATCH' ⟨*events*⟩

⟨*transformations*⟩ ::= ⟨*transformations*⟩ ⟨*transformations*⟩
| 'GROUPBY' '(' ⟨*fields*⟩ ')'
| 'FILTER' '(' ⟨*filter_matches*⟩ ')'
| 'MAP' '(' ⟨*field_expression*⟩ ',' ⟨*field_name*⟩ ')'

⟨*fields*⟩ ::= ⟨*field_name*⟩ ['.' ⟨*fields*⟩]
| 'LOCATION' ['.' ⟨*fields*⟩]

⟨*filter_matches*⟩ ::= '(' ⟨*filter_matches*⟩ ')'
| ⟨*filter_matches*⟩ '||' ⟨*filter_matches*⟩
| ⟨*filter_matches*⟩ '&&' ⟨*filter_matches*⟩
| ⟨*filter_match*⟩

⟨*filter_match*⟩ ::= ⟨*field_name*⟩ ⟨*compare_op*⟩ ⟨*field_name*⟩
| ⟨*field_name*⟩ ⟨*compare_op*⟩ ⟨*value*⟩

⟨*events*⟩ ::= '.' '@' ⟨*location_spec*⟩
| ['!'] '(' ⟨*event_match*⟩ ')' '@' ⟨*location_spec*⟩
| '(' ⟨*events*⟩ ')'
| ⟨*events*⟩ ⟨*events*⟩
| ⟨*events*⟩ ⟨*regex_op*⟩
| 'SHUFFLE' '(' ⟨*events_list*⟩ ')'
| 'CHOICE' '(' ⟨*events_list*⟩ ')'

⟨*events_list*⟩ ::= ⟨*events*⟩ [',' ⟨*events_list*⟩]

⟨*location_spec*⟩ ::= 'ANY'
| ⟨*loc_matches*⟩

⟨*loc_matches*⟩ ::= ['NOT'] '$'⟨*loc_name*⟩ [',' ⟨*loc_matches*⟩]

⟨*event_match*⟩ ::= ⟨*field_match*⟩ [',' ⟨*event_match*⟩]

⟨*field_match*⟩ ::= ⟨*terminal*⟩ ⟨*compare_op*⟩ ⟨*terminal*⟩

⟨*terminal*⟩ ::= ⟨*field_name*⟩
| ⟨*value*⟩
| '$'⟨*variable_name*⟩
| 'TIME'

Figure 5: Grammar for *Aragog*'s IV specification language. Tokens ending in '_name' are identifiers that must begin with a letter; the 'compare_op' token refers to the class of operators '==', '!=', '<', etc; 'value' indicates a constant number; and 'field_expression' is a mathematical expression over fields.

5.2.2 Event Expressions

Users define invariant violations over the transformed event streams by specifying sequences of events that result in a violation of a particular policy. Users specify these sequences with a regular-expression-like language, which describes patterns over pre-defined elements. In *Aragog*'s case, the elements take the form of a set of matching operations over the fields of the event message; the example in Figure 4 shows matches on one such field, the `eventType`. A match can occur at any point in the stream of events and triggers on every occurrence of the match, not just the first. For example, if events $A \rightarrow B \rightarrow A$ form a violation and (at runtime) we observe the sequence *CABABAC*, *Aragog* will alert twice.

As in other regular languages, users can list the sequence of expected elements and use operators like '*', '+', and '?' to signify repetitions. Users can also leverage the functions **CHOICE** and **SHUFFLE**. In **CHOICE**, an occurrence of any one of

```
1  FILTER(eventType == INIT || eventType == DROP)
2  GROUPBY(LOCATION)
3  MATCH
4    (eventType == INIT, srcIp == $S, dstIp == $D,
         srcPort == $P, dstPort == $Q) @ ANY
5    (. @ ANY)*
6    (eventType == DROP, srcIp == $D, dstIp == $S,
         srcPort == $Q, dstPort == $P) @ ANY
```

Figure 6: An example specification that checks that a stateful firewall does not drop reverse traffic for an open connection.

the contained expressions matches. In **SHUFFLE**, the contained events can arrive in any order, but must all arrive.

Event expressions come after the set of transformations and must appear after a **MATCH** statement.

Locations. In distributed NFs, an important feature is that correct behavior is defined not only on the events and their order, but on *where* the events occurred. Therefore, every event match is accompanied by a location specifiers. This is useful for specifying matches, but it is also important for determining how we might partition evaluation of the IV specification across both local and global verifiers (see Section 6). In both cases, the goal is to determine whether each pair of events are expected to occur at the same or at different NF instances.

Consider again the example in Figure 4. The example contains a single named location, `$X`, corresponding to the original primary node for the current flow. One way to use this named location is to specify that another event in the sequence must *also* occur at `$X`. Another, demonstrated in lines 7&8, is to specify that the event occurs at a location distinct from `$X`. Note that the syntax does not constrain the relationship between the locations of the events of lines 7&8.

Every event can reference one or more named locations, or it alternatively use the location `ANY`, which indicates no special semantic meaning of the location of the event. In the case of multiple locations, users specify multiple predicates (one per location). For example, to ensure three events with distinct locations: one could specify ev_1 at (`$X`, `NOT $Y`); ev_2 at (`NOT $X`, `$Y`); and ev_3 at (`NOT $X`, `NOT $Y`).

One possible method of implementing locations is to enumerate all possible locations in the system and expand the event expression accordingly. While this would allow the usage of more traditional state-machine evaluation techniques, it would also lead to an unacceptably inefficient implementation. Further, any change in membership would require us to fully recompile and re-install all IV specifications across the system. Instead, *Aragog* lazily tracks all potential candidates for location variables at runtime using a multi-leveled tree data structure, which we describe in detail in Section 6.

Variables. *Aragog* generalizes the state tracking afforded to locations in order to track other types of state in the IV specification. Examples of non-location stateful properties include the IP/port NAT mappings of the NATGW and connection tracking in a firewall. An example of the latter is shown in Figure 6, which verifies that if an outbound flow from source IP `$S` and destination IP `$D` is properly initialized, then packets in the reverse direction are also allowed.

```
1  MAP(srcIP < dstIP ? srcIP : dstIP, IP1)
2  MAP(srcIP < dstIP ? dstIP : srcIP, IP2)
3  MAP(srcIP < dstIP ? srcPort : dstPort, port1)
4  MAP(srcIP < dstIP ? dstPort : srcPort, port2)
5  FILTER(flag == FIN || flag == ACK || flag == FIN_ACK)
6  GROUPBY(IP1, IP2, port1, port2)
7  MATCH
8    (flag == FIN) @ $X
9    SHUFFLE(
10     (flag == FIN, TIME == $s) @ $Y,
11     (flag == ACK, TIME == $t) @ $Y)
12   (flag == SYN, TIME - min($s, $t) <= 30000) @ $X
```

Figure 7: An example of a timing violation specification that checks the behavior of TCP's TIME-WAIT state [22]. The SYN *must not* arrive by a deadline. This specification assumes that only packet sends are captured.

```
1  FILTER(flag == FIN || flag == FIN_ACK)
2  GROUPBY(IP1, IP2, port1, port2)
3  (eventType == FIN, TIME == $t) @ ANY
4  ((eventType != FIN_ACK, TIME - $t <= 30000) @ ANY)*
5  (TIME - $t > 30000) @ ANY
```

Figure 8: An example of a timing-related IV specification that checks timely arrival of a FIN_ACK after a FIN. The FIN_ACK *must* arrive by a deadline.

As these variables do not indicate or impose restrictions on the location of the event, we do not use them for the partitioning procedure of Section 6.

Timing. Timeouts and deadlines are also common in NFs. To specify them, users can use parameterized variables in conjunction with a builtin `TIME` field to compare the time between multiple events. For example, Figure 7 defines a violation of the TIME-WAIT semantics of a TCP flow in which SYN packets should not be sent within 30 s of a passive closer's FIN/ACK. The same SYN packet 31 s after the FIN/ACK would not be a violation. On the other end of the spectrum, Figure 8 defines a violation where a FIN-ACK does not arrive in time (within 30 s of the FIN). Any intervening FIN-ACK will mean that the violation does not match.

6 State Machine Generation

Aragog checks for invariant violations efficiently by translating each of the IV specifications into a state machine. In contrast to traditional finite-state automata, *Aragog* requires a combination of complex features, e.g., timing, arithmetic, field/location variables, and regular expression-event patterns.

Aragog, thus, generates its state machines in three stages. First, it creates a symbolic non-deterministic finite automaton (SFA) [12] whose alphabet is based around a theory of arithmetic and boolean algebra, and whose predicates can include the placeholder variables described in the previous section.

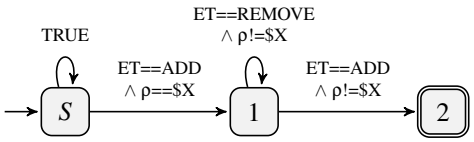


Figure 9: SFA for Figure 4 with some field names and constants abbreviated as well. ρ indicates location.

Second, it determinizes the SFA to a symbolic deterministic finite automaton (SDFA) to reduce runtime overhead of state machine execution. Finally, it constructs localized versions of the SDFA that can be used to infer the global state of the system from only locally observed events.

6.1 Constructing the SFA

We first convert all predicates on events into boolean logic with equalities/inequalities by taking the conjunction of all event field matches and the location specifier. For example, we transform an event match `(A==B, C==D) @ NOT $X` to the predicate `(A==B ∧ C==D ∧ ρ!=$X)`, where ρ is the placeholder for the event's location, which we determinize at runtime. A '`!`' modifier on the event would negate this predicate.

Aragog performs an additional check on the sequence of generated predicates to facilitate efficient variable checking (Section 7.2). Specifically, it checks via reachability analysis that all uses of variables in either an arithmetic expression or non-equality comparison ($<$, $\leq$, $>$, and $\geq$) strictly follow after their introduction via an equality comparison.

With the resulting predicates, *Aragog* constructs the SFA by creating a start state, S, with a self-loop for any event (`TRUE`). This self-loop ensures the pattern will match starting from anywhere in the event trace. From the initial state S, *Aragog* recursively builds out the state machine using Thompson's construction [38], treating `CHOICE` as a choice operator, and expanding `SHUFFLE` to all permutations. Figure 9 shows a (minimized) SFA for the example violation specification from Figure 4. We mark the final state in the SFA as the accepting state, which indicates a violation when reached.

The specified transitions may not cover the complete space of possible events. All events that do not match any transition out of the current state will never lead to a match.

Aragog next determinizes the SFA: it generates an efficiently executable DSFA from the SFA using standard symbolic automata techniques [12]. The result is a state machine where all transitions are unambiguous and exhaustive. Figure 10 shows the DSFA for the example. Each state in the DSFA stores the correesponding set of SFA states the machine is in at that given point in time.

6.2 Local State Machines

Conceptually, the DSFA provides an efficient method for checking whether a stream of events leads to an invariant violation. In principle, we could simply funnel all events to a central verifier, which would then apply the relevant DSFA transition and report a violations upon reaching an accepting state. Unfortunately, doing so would require the verifier to process *all* unfiltered events in the system. Instead, we further improve *Aragog*'s scalability by generating a localized version of the state machine that is executed on the same machine as the NF before sending the event to the global verifier.

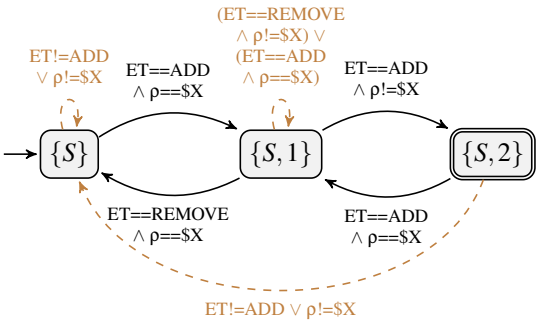


Figure 10: DSFA for the SFA in Figure 4. Colored, dashed edges represent suppressible transitions.

6.2.1 Suppressible Transitions

The local state machine needs to identify events that will not impact the detection (or lack of detection) of a user-specified violation whether or not it is sent to the global verifier. Our key observation is that there are transitions in the global DSFA that do not affect the end result of the state machine. We term these transitions *suppressible transitions*. More formally:

Definition 1. An event stream s is either empty $s = \varepsilon$ or it consists of an event followed by another stream $s = e \cdot s'$.

Definition 2. $q \xrightarrow{e} q'$ indicates that, from state q, event e transitions to state q'. We lift this to event streams inductively as $q \xrightarrow{\varepsilon} q$, and $q \xrightarrow{e \cdot s} q''$ iff $q \xrightarrow{e} q'$ and $q' \xrightarrow{s} q''$.

Definition 3. Transition t is suppressible if for any event e matching t from state q, then (1) $q \xrightarrow{e} q'$ means q' is not an accepting state, and (2) for any event stream s, and accepting state q_a then $q \xrightarrow{e \cdot s} q_a$ iff $q \xrightarrow{s} q_a$.

In the running example DSFA in Figure 10, the three dashed transitions are suppressible given the above definition. The two self-loops are clearly suppressible (satisfy Definition 3) since an event processed by such a loop will not change the global state—(not) observing the event has no effect, and the loops do not occur on accepting states. Perhaps less obvious is that the bottom-most edge is also suppressible since, from either state $\{S\}$ or $\{S,2\}$, one needs to see the same two events to get back to the accepting state $\{S,2\}$. For example, an `ADD` event at `$X` followed by another at `NOT $X` will take either state $\{S\}$ or $\{S,2\}$ back to $\{S,2\}$. We never mark transitions with time constraints as suppressible—we assume the timing of an otherwise irrelevant event might still be significant.

Algorithm 1 Create a local state machine for a variable

```
1:  input: Global DSFA G, variable V, filter F
2:  output: Local DSFA L
3:  procedure CreateLocalDFA(G, V, F)
4:      L := CopyStates(G)
5:      for S ← States(G) do
6:          for T ← Transitions(G, S) do
7:              P := Predicate(G, T)
8:              if SAT((F ∧ P) ⇏ (ρ = V)) then
9:                  AddTransition(L, TargetState(T), ε)
10:             P′ := Simplify(P, ρ==V)
11:             AddTransition(L, TargetState(T), P′)
12:     return Determinize(L)
```

6.2.2 Local State Machine Construction

Aragog uses local knowledge to determine whether an event will be processed by a suppressible transition. Since each local component is unaware of what might be happening at other components, it must conservatively account for all possibilities. To determine (locally) whether an event is suppressible, we create a local state machine for every location variable in every IV specification such that each machine assumes it is playing the role of that location (*e.g.*, one machine for "I might be `$X` in a violation" and another for "I might be `$Y` in a violation"). In the example from Figure 10, there is only a single local state machine: the one for `$X`.

The first step in creating a local state machine, L, is to model the uncertainty other locations may introduce (Algorithm 1). The algorithm takes the global state machine G, the location variable V (*e.g.*, `$X`), and a predicate F corresponding to the user-defined **`FILTER`** statements. It returns a new localized SDFA.

The algorithm considers each transition T in G where T has predicate P, and checks whether the formula $(F \wedge P) \not\Rightarrow (\rho = V)$ is satisfiable (line 8). If it is, then there exists a potential event that makes it through the filter F and uses transition T but which takes place at a location other than V. To model the fact that other NF instances might send events that use this transition, the algorithm adds to L an epsilon (ε) transition (line 9). An ε transition is one which the local SFA can take immediately and unconditionally. It accounts for the possibility of concurrent execution of other NF instances to represent that the global state could be in either state (the one before or the one after the ε transition).

In either case, the algorithm then adds a local transition to L by simplifying the existing transition predicate (P) to account for the fact that the location is known (line 11). It does so by partially evaluating the predicate with the assumption that ρ==V (line 10). In Figure 10, for example, the transition (`ET==REMOVE` ∧ ρ==`$X`) is simplified to `ET==REMOVE`.

Figure 11 shows the local SFA for location `$X` and its determinized (DSFA) form. By executing the DSFA in Figure 11 locally, an NF instance can learn some partial information about the state of the overall system. For example, after seeing

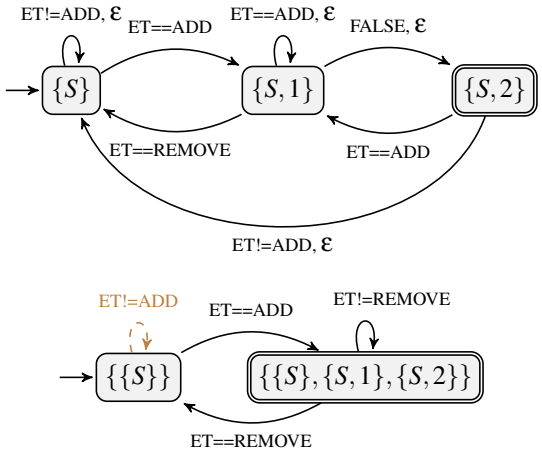


Figure 11: Local machine for `$X` from Figure 10. SFA is shown on top and its equivalent DSFA is shown below. Colored, dashed edges indicate locally suppressible transitions.

an `ADD` event, the NF instance recognizes that (if it is `$X`) the global state machine can be in any state: $\{S\}$, $\{S,1\}$, or $\{S,2\}$. However, after locally processing a `REMOVE` event, the local machine now knows it must be in state $\{S\}$ once more.

6.2.3 Suppressing Events Locally

The local machine can hide events when it can prove they would otherwise be processed by suppressible transitions in the global machine. Algorithm 2 is used to create all the data structures needed to suppress events locally. It takes the global state machine G as input along with the user-defined filters F and produces, as output, a collection of local state machines (L_i) as well as a negated condition (NC), explained below.

The algorithm works by iterating over every location or variable in the IV specification (line 5) and calling CreateLocalDFA to build the local state machine (line 6). It then walks over each local transition (T) and attempts to mark the transition as locally suppressible. To do so, it looks up all the possible global states corresponding to this local state (line 11) and checks whether the local transition can process an event that is also processed by, and is not suppressible for, some global transition T′ from one of these states (line 16). If not, then all events that trigger T must be part of a suppressible transition in the global DSFA, so the event is suppressed.

In Figure 11, events matching `ET!=ADD` in state $\{\{S\}\}$ are suppressible: for each global state in the set ($\{S\}$), this event must be processed by a suppressible global transition.

Negated condition. The final part of the algorithm (lines 20 to 23) computes a "negated condition." This condition captures the case where the local NF may not correspond to any named location in the IV specification, *e.g.*, the NF instance is not `$X`, but it still may observe a relevant event as `NOT $X`. We observe, in such a case, the current machine can not possibly know anything about the global automaton

Algorithm 2 Construct local state machines

```
1:  input: Global DSFA G, filter F
2:  output: Local state Θ = ⟨{L_1, ..., L_k}, NC⟩
3:  procedure LOCALIZE(G, F)
4:      NC := false, LS := ∅
5:      for V ← Variables(G) do
6:          L := CreateLocalDFA(G, V, F)
7:          for S ← States(L) do
8:              for T ← Transitions(L, S) do
9:                  suppress := true
10:                 P := Predicate(L, T)
11:                 for S′ ← GlobalStates(L, S) do
12:                     for T′ ← Transitions(G, S′) do
13:                         if CanSuppress(G, T′) then
14:                             continue
15:                         P′ := Predicate(G, T′)
16:                         if SAT(P ∧ (ρ = V) ∧ P′) then
17:                             suppress := false
18:                 if suppress then MarkSuppressed(L, T)
19:         LS := LS ∪ {L}
20:     for S′ ← States(G) do
21:         for T′ ← Transitions(G, S′) do
22:             if CanSuppress(G, T′) then continue
23:             NC := NC ∨ Simplify(Predicate(G, T′), ρ==Fresh())
24:     return ⟨LS, NC⟩
```

state since the other NF instances that also are not `$X` may be sending events that match `NOT $X` transitions. The fix is simple: the algorithm computes the disjunction of all the transition predicates in the global state machine subject to the knowledge that the location ρ does not match any variable (line 23).

In the running example, the algorithm computes: (`ET==ADD` ∧ `Z==$X`) ∨ (`ET==ADD` ∧ `Z!=$X`) ∨ (`ET==REMOVE` ∧ `Z==$X`), where `Z` is a fresh variable that is guaranteed to not match any location in the predicate. The above condition simplifies to **`ET==ADD`**. This means that the local machine *must* send any `FLOWCACHE_PRIMARY_ADD` events to the global verifier regardless of its local state.

Note that non-location variables may introduce some uncertainty at the local verifier, which may not be sure what other NF instances have observed for their value. To address this, *Aragog* first tries to generate a predicate that accounts for any possible variable assignment by enumerating all possible assignments from their ==/!= expressions, replacing their occurrences in the negated condition, and computing the disjunction of the resulting predicates. If any variables or arithmetic operations remain in the disjunction, *Aragog* will simply not suppress any events, which is always safe.

7 Runtime System

We next describe the *Aragog* runtime.

7.1 Workflow Overview

We begin with the common case: NF instances synchronized via PTP send events—at runtime—to a co-located local agent via traditional IPC mechanisms. This local agent applies transformations, computes supressions using local state machines, and then sends any non-suppressible events to the global verifier via a set of Kafka brokers.

Filtering, mapping, and grouping. After ingesting the stream of PTP-timestamped events, local *Aragog* agents co-located with the NF first apply any applicable transformations—**`FILTER`**, **`MAP`** or **`GROUPBY`**—to the raw stream. As each IV specification can have a different set of transformations, this may require *Aragog* to duplicate the incoming stream of raw events (it tries to avoid doing so when possible). The end result is a set of keyed event streams: one stream for each combination of policy and **`GROUPBY`** key.

Computing suppression. The next step, also performed locally, is to determine whether events in each keyed stream are suppressible. *Aragog* passes the events through the localized state machines — one for each location referenced in each IV specification. For a given event and IV, *Aragog* suppresses the event when (1) all localized instances of the IV specification would take a suppressible transition when fed the current event and (2) the event does not satisfy the negated condition. If either constraint is false, *Aragog* sends the event to a Kafka queue for the given keyed event stream.

As a concrete example, Figure 12 shows processing of a series of events with the specification in Figure 4 and with the same **`GROUPBY`** key. The first event is an `ADD` event at flow decider FD_1. After seeing this event, FD_1 will transition locally from state q_0 ($\{S\}$) to state q_1 ($\{\{S\},\{S,1\},\{S,2\}\}$). Since this transition is not suppressible, the event is sent to the verifier. The next event is a `REMOVE` event that takes place at FD_3. This particular transition *is* suppressible and the negated condition (`ET==ADD`) is not satisfied, thus, the event is suppressed.

This suppression can substantially reduce the number of events received by the global verifier. For example, with three replicas (including the primary), a correct execution of Figure 4 *Aragog* would receive—after suppression—just 2 out of 4 events (the add and remove at the primary but not the 2 suppressed removes at nodes other than `$X`).

Global state machines. Pulling from Kafka is a cluster of Flink instances running the global versions of the IV state machines. Both the Kafka and Flink instances are automatically provisioned, checkpointed, assigned **`GROUPBY`** keys, and load balanced to worker nodes. As Flink does not guarantee that events from different NF instances will arrive in order, *Aragog* temporarily stores and reorders events in the Flink workers with an efficient priority queue before passing them to the associated state machine.

One challenge is how long to wait for delayed events. One approach is to maintain a list of all NF instances along with the timestamp of the last event they sent to this partition and only process time t when we have seen events from all instances up to $t + latency$. Unfortunately, most NF instances do not interact with most flows/policies and sending 'null'

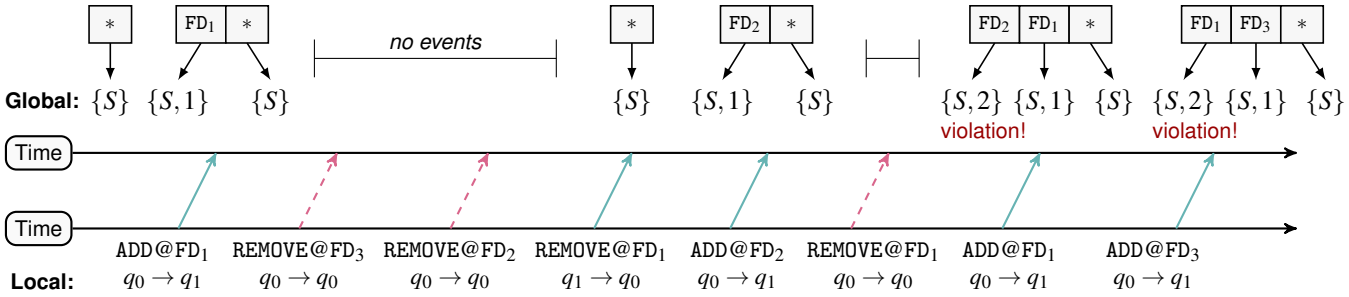


Figure 12: Distributed execution for the example from Figure 4 on an example sequence of events for N flow deciders. Time progresses from left to right. Local events are shown along the bottom line with the local state of the flow decider. We use $q_0 = \{\{S\}\}$ and $q_1 = \{\{S\},\{S,1\},\{S,2\}\}$. The global verifier's state is shown at the top. Red, dashed edges indicate suppressed events.

events to advance the timestamps of every partition would be costly. Instead, *Aragog* relies on the assumption of a maximum latency t_{max} and handles violations of this assumption with the techniques in Section 7.3.

Aragog will hold each event for t_{max} time before running it through the global DSFA. While processing events for a given IV specification, the verifiers will track all of the possible states in which the associated state machine could be, as well as all potential values of the IV specification's variables (see Section 7.2 for details). If any of the possible states is a 'final' state in the IV's DSFA, *Aragog* will raise an alert.

Consistent sampling. If scaling is still challenging despite sharding the verifier, filtering relevant events, and suppressing events locally, *Aragog* provides a final mechanism that lets users trade performance for completeness by sampling a consistent set of events with consistent hashing based on the `GROUPBY` key (*e.g.*, a 5-tuple for NATGW). In this way, each group is itself complete though false negatives remain possible when violations occur for keys that are not sampled.

7.2 (Location) Variable Tracking

Aragog tracks *all* possible instantiations of variables (location or otherwise) at runtime using a multi-level tree data structure (shown at the top of Figure 12). Intuitively, the tree captures the state the global automaton would be in for every possible instantiation, with the leaves of the tree as the state and the interior nodes as variable assignments. Every variable is assigned a single level of the tree.

Let the number of variables (location or otherwise) for an IV specification be n. When the system starts, the DSFA is in the start state, $\{S\}$, for all possible variable assignments. This is represented as a degenerate tree with height $n+1$ and a single leaf pointing at the start state $\{S\}$. The interior nodes are all set to $*$, indicating no constraints on the n variables. For every incoming event, we advance the DSFA using the state and variable assignments of every leaf. Whenever a predicate is encountered that references a variable, V_i, if $V_i = *$ is an ancestor of the current leaf we split execution into a case where V_i satisfies the predicate and a case where it does not. The $(n-i)$-height subtree under $V_i = *$ may need to be cloned.

In the example of Figure 12, there is only one variable (`$X`) and, thus, only two levels in the tree. The system starts in the degenerate case where `$X` $= *$. After the first `ADD` event arrives at the verifier from $\texttt{FD}_1$, we fork the tree to separate out the old case and a new case for `$X`=$\texttt{FD}_1$. When `$X` is $\texttt{FD}_1$, the verifier takes the transition (`ET` $==$ `ADD` $\wedge$ ρ $==$ `$X`) to state $\{S,1\}$: the current location ρ is $\texttt{FD}_1$, and `$X` is also $\texttt{FD}_1$. Otherwise if `$X`!=$\texttt{FD}_1$, it takes the self-loop transition to remain in $\{S\}$. For the next event from $\texttt{FD}_1$ (`REMOVE`), there is no new case to fork, and applying the transition to both cases in the tree leads to both being in state $\{S\}$ once more. Therefore, the states are collapsed together back to $*$. This process continues until the second to last event where a violation is detected for the case where `$X` $= \texttt{FD}_2$ due to a duplicate add at $\texttt{FD}_1$. The final event (`ADD` at $\texttt{FD}_3$) leads to a second violation, where now `$X` $= \texttt{FD}_1$, and is subsequently caught by the implementation.

7.3 Fault Tolerance

Failures and message drops/delays can cause *Aragog* to become desynchronized from the ground-truth state of the system. Even so, *Aragog* is able to guarantee both precision and recall of typical network violations under the assumption of 'partial synchrony' [15], *i.e.*, that there exists a time, t_s, after which there is some upper bound on message delivery time.

- *Recall:* Under a partial synchrony assumption, *Aragog*'s practice of creating a self loop in the initial state of the SFA means all violations whose trace begins *after* t_s are accurately modelled in the state machine and detected.
- *Precision: Aragog*'s precision guarantees are less complete, but still hold in practice. Specifically, we observe that all of the IV specifications we studied contained some property where flow state would eventually be dropped in reaction to a `REMOVE_ENTRY` or TCP `FIN`/`RST` event; such transitions are common in networked systems and ensure that any desynchronized state machine instances will eventually transition back to the initial state.

In addition to the above, Flink provides guarantees that successfully pulled events are processed by the state machine

Network Function	Invariant Description	LoC	States	Transitions
NAT Gateway	`nat_decider_open`: After a PW goes into closed state, at least one replica also goes into closed state.	14	4	10
	`nat_consensus`: All TCP flows are open only after consensus.	5	2	4
	`nat_open_to`: Open flows are timed out after 4 minutes of inactivity.	5	4	12
	`nat_primary_single`: There is a single primary per flow.	10	3	7
	`nat_primary_to`: The NATGW does not start an idle timeout for active flows.	13	6	18
	`nat_same_consensus`: After TCP flow U is terminated, the next flow for U achieves consensus.	12	5	15
	`nat_syn_to`: Flows with a TCP handshake in progress timeout after 5 seconds of inactivity.	5	4	12
	`nat_udp_same_consensus`: If UDP flow U times out, the next flow for U achieves consensus.	12	6	17
Firewall [5]	`fw_consistency`: *all* Firewall instances should block suspicious IPs after a block rule is added.	6	4	12
	`fw_client_init`: Ensure a flow can only be open after a client initiates it.	4	2	4
	`fw_syn_first`: Data packets are only allowed after a SYN is sent.	4	2	4
DHCP	`dhcp_reuse`: Leased addresses are not re-used until expiration or release.	6	4	12
	`dhcp_overlap`: Leases should not overlap between DHCP servers.	6	3	7

Table 1: List of example invariants that *Aragog* can implement for several common network functions and systems.

exactly once. *End-to-end* guarantees of exactly once delivery between Flink and Kafka are also possible, but would incur the overhead of atomic exporting of NF events, transactions, and rollbacks. Instead, *Aragog* chooses to rely on partial synchrony and to alert users after the fact when false positives may have occurred. This can happen when an event arrives with a timestamp earlier than the last processed event, two events arrive from an NF instance with a gap in their sequence numbers, or an NF instance (and its local agent) fail. Upon restarting, the agent can immediately resume exporting events, but the local state machine may be out of sync. In this case, it can temporarily export all events (which is always safe) until it can synchronize with the global verifier to rebuild the local state machines from the global verifier's state.

8 Implementation

We have implemented *Aragog* with more than 6,500 lines of Java 8 code, packaged with Maven v3.6 and more than 2,000 lines of C++ code. The implementation consists of two major components: the compiler and runtime system. It can be found at: `https://github.com/microsoft/aragog`.

The compiler takes as inputs an event format specification as described in Section 5.1 along with a set of IV specifications in the format of Section 5.2. For each IV specification, it generates the global state machine, the resulting local state machines, information about suppressible events, and a slew of other metadata about variables, filters, and partitioning. The lexer and parser use the ANTLR v4.7 [1] parser generator, and the SFA construction and determinization use the open-source `symbolicautomata` library [6], but with the addition of a custom Z3-based [7] theory of Boolean Algebra designed to support our IV specification language.

We built the runtime system on top of Apache Flink [2] and Kafka [3]. These frameworks are designed for scalable and robust stream processing and provide, intrinsically, fault-tolerant and stateful processing, exactly-once semantics, load balancing, flexible membership, checkpointing, etc. The local agents, implemented in C++, ingest events directly, then filter, map, and suppress events as necessary before sending them to Kafka. The global verifiers, implemented in Java using Apache Flink, pull from Kafka into a timestamp-based priority queue from which events are dequeued after waiting for a maximum delay; violations are logged to disk. We place the verifiers off of the critical path to avoid any impact on production traffic.

9 Evaluation

We evaluate *Aragog* in CloudLab [37] with a number of network functions and along a number of dimensions.

The deployed NAT gateway (§2). We use two event traces captured from two different builds of the NAT gateway to evaluate *Aragog*. The builds capture the introduction of a set of bugs that arose from the change of an interface between two internal components, with V1 from before the change and V2 from after. The traces are both for 7 flow deciders over a 30 minute interval, but they export a different number of packets (V1: 23.7M; V2: 9.0M) owing to changes in the protocol. The production deployment of NATGW does not yet support fine-grained clock synchronization, but our operators plan to add it in the system's next version. Instead, we capture the event traces and correct for time drift using a set of known synchronization points within the event stream. In total, there are eight IV specifications for NATGW (see Table 1).

A distributed firewall. We also execute a collection of micro-benchmarks using an open-source, stateful, and distributed firewall implementation built on `iptables`, `conntrackd`, and `keepalived` [5]). On the firewall, we check various invariant violations, some of which were derived from [8]. The list of specific invariant violations we check are listed in Table 1.

We deploy this firewall on a topology with four clients, four internal hosts on a single LAN, and four firewall nodes interposing between the two groups. The firewalls are configured as two high-availability groups with one primary and one hot standby each. Each primary-standby group shares a virtual IP with the VRRP protocol. We base the traffic between external hosts and internal servers on the traces provided in [9].

Invariant Violation	Version 1	Version 2
nat_decider_open	0	0
nat_consensus	0	0
nat_open_to	1	45019
nat_primary_single	0	0
nat_primary_to	1	29964
nat_same_consensus	536	259
nat_syn_to	0	2697
nat_udp_same_consensus	0	0

Table 2: Violations found in traces for NATGW versions. Note that V1's trace contains more events than V2's, which may account for the difference in `nat_same_consensus` violations.

DHCP. To show the flexibility of *Aragog* and its language, we also give examples of DHCP invariant violations in Table 1. With our current implementation, the operator needs to write just 6 lines to express the invariant violations. Each of the state machines uses a small number of states and transitions.

Evaluation metrics. We evaluate *Aragog* along a number of key dimensions: lines of code, throughput, latency, and CPU overhead. In addition, our micro-benchmarks show *Aragog*'s ability to scale as the number of nodes in the NF deployment increase by demonstrating the benefits of our event suppression scheme. Finally, we find *Aragog* is able to identify bugs in production systems. In particular, we were able to identify four bugs in the NAT gateway which were confirmed by our operators. Similarly, in the firewall, *Aragog* was able to find a series of injected configuration errors over real traffic traces.

9.1 Bugs Identified by *Aragog*

NATGW Bugs. Running the traces through *Aragog*, we discovered violations of `nat_open_to`, `nat_primary_to`, `nat_same_consensus`, `nat_syn_to`, all of which were confirmed as caused by bugs by the NATGW team. Table 2 shows the absolute number of violations observed for each.

`nat_open_to` was by far the most frequent violator in V2. Discussions with our operators revealed that in V2, this violation (and that of `nat_syn_to`) were due to related bugs in the code: it had taken operators over an hour to identify the issues while *Aragog* identified it in under a minute. Although `nat_open_to` also had a violation in V1, further examination revealed that the violation in V1 was due to an expected consequence of eventual consistency—specifically one of the replicas was getting update messages from the packet worker but the primary did not and therefore started a timeout for the flow. This led us to start checking for `nat_primary_to`.

Also prominent in both systems were violations of `nat_same_consensus`. This violation occurred because the flow was not closed or removed properly from one of the replicas. The operators suspected this could be an issue, but never had a method to test that hypothesis. *Aragog* confirmed the problem and helped the developers to formulate the test setup to reproduce the issue.

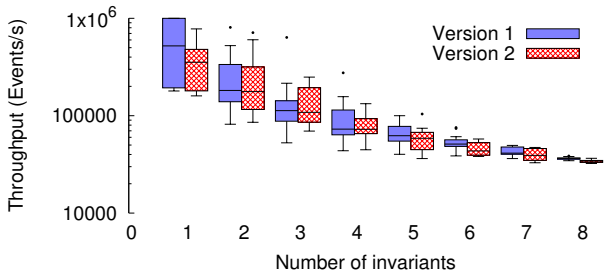


Figure 13: The throughput in events/second for an executor of *Aragog* on the trace.

Bugs in the distributed firewall rules. For the firewall, we manually injected bugs in the firewall configuration to test *Aragog*'s ability to identify this category of errors. The injected issues, for instance, always allowed external traffic from a particular address range into the internal network, violating `fw_client_init`. *Aragog* found all of them.

9.2 Throughput of *Aragog*

Aragog's global verifier keeps track of the set of possible states for each IV specification and the possible values for each variable/location. Thus, *Aragog*'s throughput is directly correlated with the number of IVs checked (Figure 13). To evaluate this scaling, we run the V1/2 traces through all the 8 NATGW IV specifications using a single Task Slot on the global verifier (running on an Intel(R) Xeon(R) E5-2450 processor CPU @ 2.10GHz machine). We upload the entire trace on Apache Kafka after local processing to measure the maximum throughput a single task slot of Apache Flink of the global verifier can process. In Figure 13 we randomly select n among the NATGW invariant violations and see the performance. As each type of invariant violation exhibits different resource requirements, we see more variance when the number of type of invariant violations selected is low.

With a single task slot, our optimizations allow *Aragog* to scale and process over 500,000 events per second for a single invariant violation type (over 30,000 for 8). Adding more task slots does not improve the performance as our implementation is parallel in nature and a single task slot is already using multiples core in a single machine.

Aragog scales linearly as we add more machines to the global verifier (Figure 14). Scaling with multiple machines avoids the bottleneck of CPU and I/O.

9.3 Overhead of *Aragog*

To measure the memory and CPU overhead of *Aragog*, we study its behavior while verifying the distributed firewall. In Figures 15, 16 and 17, data is divided into separate groups. 'Primary' represents the verifier running at the primary firewall. 'Backup' represents the verifier running at the hot-standby firewall. 'Manager' and 'executor' represent the Apache Flink job manager and executors, respectively. The global verifier runs on the executors.

Process/Location	Resource	Spearman correlation
job manager	CPU	0.14700
job manager	memory	−0.59379
executor	CPU	0.78481
executor	memory	−0.38373
primary	CPU	0.88916
primary	memory	−0.18253
backup	CPU	0.93618
backup	memory	0.24768

Table 3: Spearman Correlation between number of events/s and resource utilization at different locations of verifier while running the firewall.

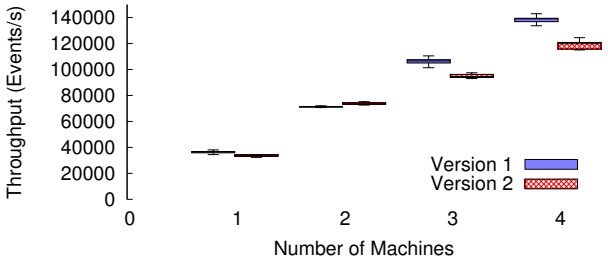


Figure 14: Throughput of multiple *Aragog* verification server checking all 8 types invariant violations

We see that in Figures 16 and 17, the overhead of the local verifiers is low. This is important as the local components are co-located with the production NF instances. To that end, the CPU utilization of the local verifier increases linearly with the number of flow events per second. We also observe the CPU and memory usage for the local verifier is higher at the primaries as they tend to generate more events. Memory at the local components is much less correlated (Table 3), partly due to *Aragog*'s small memory footprint (Figure 17).

The global verifier has higher CPU (Figure 15) and memory (Figure 17) than local verifiers as the global verifier is implemented in Java using Apache Flink. We have set the maximum memory of job manager to 1 GB and executor to 2 GB. In our graphs, we are plotting active memory in Java's heap for the global verifier rather than used memory to avoid including memory waiting to be cleaned up by the Java GC.

Figure 18 shows the CDF of *Aragog*'s *time to detection* for violations in the distributed firewall function. The time to detection is low: in the median it takes roughly 70 ms from the time the event was executed (the violation occurred) at the NF instance until *Aragog* raises an alert.

9.4 Efficacy of Suppression

Each optimization in *Aragog* improves scalability by reducing the number of events sent to the global verifier (reducing the network overhead and the number of events processed at the global verifier). Filters remove the need to send events that are not pertinent and reduce the number of events sent to the verifier by up to 61% for the NATGW (Table 4). Suppressible

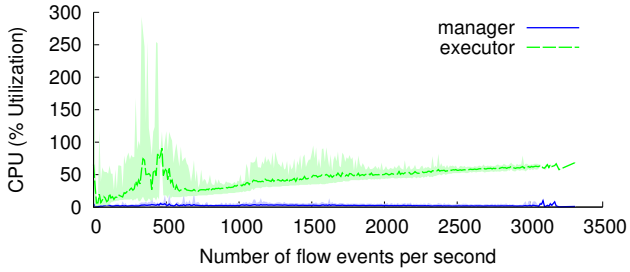


Figure 15: CPU utilization by *Aragog*'s global component. 'Manager' and 'executor' refer to the Flink node designations.

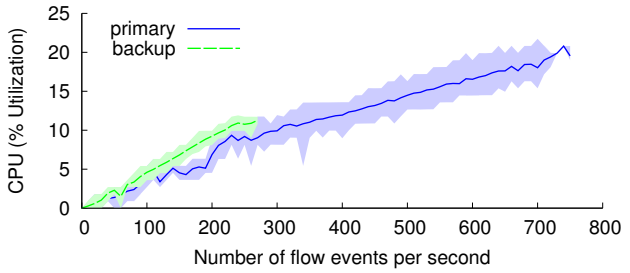


Figure 16: CPU utilization by *Aragog*'s local component. The graph shows CPU utilization of the local verifier at both the primary and backup firewall.

events can further reduce this number (by up to an additional 12% in our experiments).

10 Related Work

Runtime verification. Researchers have studied runtime verification extensively, with many papers dedicated to improving its expressiveness and performance. We find that, unfortunately, these existing systems are a poor fit for our setting. For example, D^{3}S [28] is a runtime verifier. Like *Aragog*, it focuses on identifying bugs in distributed systems at runtime, and its usage of C++ implementations to specify general-purpose properties means that it can check a wider range of properties than *Aragog*. On the other hand, *Aragog* is able to leverage its domain-specific IV specification language (based on regular expressions) to reduce overhead (*e.g.*, with event suppression). Similarly, while CrystalBall [39] can proactively steer a distributed system away from bad states, it imposes restrictions on the target system's architecture that make sense for a distributed system, but not necessarily for a large-scale NF. A third system, Pivot Tracing [31] tracks only causal relationships and not unrelated events at different machines—a property required by some of NATGW's uniqueness invariants. We emphasize that none of the above implies strict superiority. In particular, as *Aragog* is domain-customized for NFs, it should not be used for more general cases (*e.g.*, it may not be able to verify systems like Chord or Paxos efficiently).

We also note that *Aragog* borrows ideas from two areas within runtime verification. The first is verification of distributed systems, which is broadly separated into two categories based on whether the system assumes a synchronized

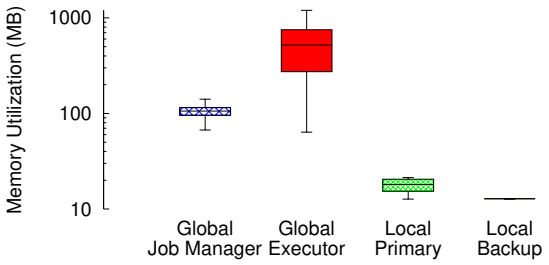


Figure 17: Memory utilization of verifier in MBytes.

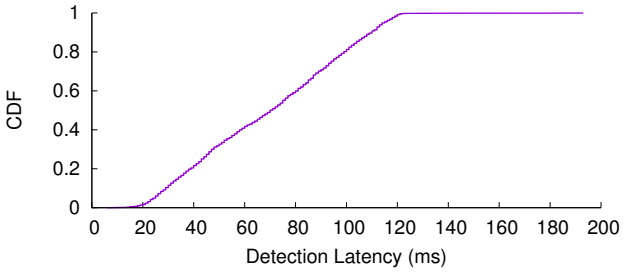


Figure 18: Latency (alert time – packet time) for detecting a violation in the distributed firewall.

global clock [17]. In this respect, *Aragog* would be considered a *decentralized* [14, 17] runtime verification system. The second is parametric verification, which focuses on checking universally or existentially quantified expressions [13, 20, 30, 36]. The location variables in *Aragog* are examples of parametric variables. The main distinction of *Aragog* from these systems is its combination of parametric and decentralized runtime verification through its support for location variables. Moreover, *Aragog*'s efficient implementation of this combination of features through its use of sharding and local symbolic state machine partitioning is new in this context.

Static verification of NFs and distributed systems. Static verification has as equally rich history, including in the domain of NFs and distributed systems [10, 34, 42, 44]. Static verification approaches may provide exhaustive guarantees of correctness, but often suffer from issues of scalability. For this reason, many static verifiers (*e.g.*, [42, 44]) assume single-machine middleboxes, while others (*e.g.*, [25, 29, 40]) may require checking an exponential number of states/paths. Leveraging hand-written NF models can improve scalability compared to verifying source code, but requires tedious and error-prone manual translation of NF models and divorces the verifier from the behavior of the actual deployed system [10, 34].

Aragog makes a different set of tradeoffs, opting to sacrifice principled exploration for improved scalability and giving up the ability to catch bugs early for the ability to test real implementations running over live data. We argue that these tradeoffs are a better fit for our operators' requirements.

Related to the above approaches is the use of semi-automated theorem provers such as Dafny [27]. Users can apply these tools to build systems that are provably correct. A good example of this approach is IronFleet [21], which was used to build a verified, Paxos-based replicated-state-machine library. On the other hand, a drawback of this approach is that it requires significant development effort. IronFleet verification, for example, involved tens of thousands of lines of proof. In contrast, *Aragog* aims to be a lightweight (but sans proof-of-correctness) alternative, requiring little to no developer effort by catching bugs at run time.

Version	Generated	After Filter	After Suppression
V1	189M	92.9M (**49.1%**)	70.2M (**37.1%**)
V2	72.2M	36.7M (**50.8%**)	28.0M (**38.8%**)

Table 4: Total number of generated events, events processed after filtering, and events processed after filtering and suppression for the NAT gateway with all 8 IV specifications.

Stateless dataplane verification. Dataplane verification tools such a HSA [23] and Anteater [32] verify the correctness of a static snapshot of network forwarding tables. Later tools such as Veriflow [24] perform runtime verification by constantly re-verifying the network state as changes occur. Each of these tools reasons about all packet behaviors—a challenging task—however, their reasoning is limited to verification of *stateless* network forwarding. In contrast, *Aragog* focuses on verifying complex temporal and stateful properties of general-purpose distributed NFs. For example, *Aragog* can ensure a stateful firewall correctly allows traffic only for connections that are established by an internal sender.

11 Discussion and Conclusion

Aragog is a lightweight verification framework for verifying distributed network functions. To scale to large systems with minimal overhead, *Aragog* leverages a two-tiered setup with local monitors at each NF instance sending events to (and hiding events from) a collection of sharded global verifiers. While *Aragog* can verify any distributed system, its scalability will depend on whether the invariant violations of interest can utilize its sharding and suppression optimization effectively.

Finally, as *Aragog* is the first to verify distributed network functions at scale (and at runtime), there are a number of aspects where follow up work may be needed. Included in this set are explorations of other time synchronization protocols, e.g., [18] or some other lightweight and precise event ordering mechanisms. Also for future work are innovations in atomic event export and transactions over streams in *Aragog*.

Acknowledgments

We gratefully acknowledge our shepherd Xi Wang and the anonymous OSDI reviewers for all of their thoughtful reviews, comments, and time. The authors would also like to thank Geoff Outhred for his feedback and support of this work. This work was funded in part by NSF grant CNS-1845749, DARPA contract HR0011-17-C0047, and a Microsoft internship.

References

[1] Antlr. https://www.antlr.org/.

[2] Apache Flink: Stateful computations over data streams. https://flink.apache.org/.

[3] Apache Kafka. https://kafka.apache.org/.

[4] Maglev outage. https://status.cloud.google.com/incident/cloud-networking/18013.

[5] NetFilter. http://conntrack-tools.netfilter.org/.

[6] A symbolic automata library. https://github.com/lorisdanto/symbolicautomata.

[7] Z3. https://github.com/Z3Prover/z3.

[8] Ehab Al-Shaer, Hazem Hamed, Raouf Boutaba, and Masum Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE journal on selected areas in communications*, 23(10):2069–2084, 2005.

[9] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[10] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Abstract interpretation of stateful networks, 2017.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[12] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification, 29th International Conference (CAV '17)*, July 2017.

[13] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 341–356. Springer Berlin Heidelberg, 2014.

[14] M. Ali Dorosty, Fathiyeh Faghih, and Ehsan Khamespanah. Decentralized runtime verification for LTL properties using global clock, 2019.

[15] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[16] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, pages 523–535, 2016.

[17] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. *Runtime Verification for Decentralised and Distributed Systems*, pages 176–210. 2018.

[18] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, pages 81–94, 2018.

[19] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72, 2016.

[20] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. *Monitoring Events that Carry Data*, pages 61–102. 2018.

[21] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob Lorch, Bryan Parno, Michael Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Communications of the ACM*, 60:83–92, 06 2017.

[22] Information Sciences Institute. Transmission Control Protocol. RFC 793, RFC Editor, September 1981.

[23] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*, pages 9–9, Berkeley, CA, USA, 2012.

[24] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, September 2012.

[25] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI '07)*, Cambridge, MA, April 2007.

[26] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[27] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010.

[28] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^3S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, page 423–437, USA, 2008.

[29] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*, New York, NY, USA, 2019.

[30] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 285–300, 2014.

[31] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic causal monitoring for distributed systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016.

[32] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 290–301, New York, NY, USA, 2011.

[33] Tim Nelson, Nicholas DeMarinis, Timothy Adam Hoff, Rodrigo Fonseca, and Shriram Krishnamurthi. Switches are monitors too! stateful property monitoring as a switch design criterion. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*, page 99–105, New York, NY, USA, 2016.

[34] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 699–718, Boston, MA, March 2017.

[35] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 207–218, 2013.

[36] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. MarQ: Monitoring at runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 596–610, 2015.

[37] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *;login:, the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[38] Guangming Xing. Minimized thompson NFA. *International Journal of Computer Mathematics*, 81:1097 – 1106, 2004.

[39] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, page 229–244, USA, 2009.

[40] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, page 213–228, USA, 2009.

[41] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. NetSMC: A custom symbolic model checker for stateful network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 181–200, February 2020.

[42] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, page 275–290, New York, NY, USA, 2019.

[43] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified NAT. In *Proceedings of the ACM SIGCOMM 2017 Conference*, page 141–154, 2017.

[44] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 221–239, Santa Clara, CA, February 2020.

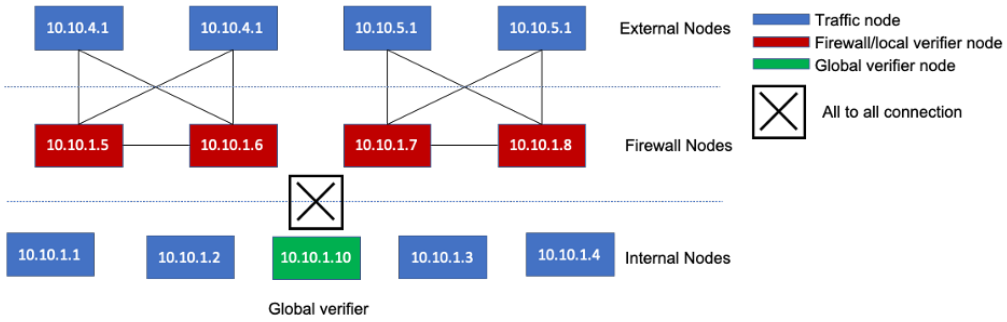


Figure 19: Topology for the distributed firewall demo.

A Artifact Appendix

Aragog is available at: https://github.com/microsoft/aragog. Instructions for installing and running the artifact can be found in the README of this repository.

A.1 Code Structure

A.1.1 SFA generation

SFA generation has three dependencies: `symbolic automata`, `z3` and `antlr`. The primary classes are:

GenerateSFA.java: This is the main class. It takes the event definition file and the IV specification file, and it outputs the SFA in a form that can be accepted by the runtime verifiers. `Antlr` is used to create the parse tree, which is then used to create the global SFA using the class `InvariantVisitor`, which recursively visits each node of the parse tree while constructing the SFA. A DSFA is generated from this automata and printed to a `.sm.g` file along with a DOT file representation.

GenerateLocalSFA.java: This class is called by GenerateSFA to create local versions of the global SFA. Specifically, it takes the SFA and locations as input and outputs the local SFA for each location. The end result of this step is a series of `.sm.[1-9][0-9]*` files, one series for each IV specification.

EventSolver: This class contains the theory of `BooleanAlgebra` logic required to create the SFA. Please refer to Section 6.1 of the paper for details.

A.1.2 Global Verifier

The global verifier has three dependencies: Apache Flink, Apache Kafka, and `antlr`. The primary classes are:

Verifier.java: This is the main class. The program creates state machines according to the provided `.sm.g` files and processes them. The input event messages can come either from a file, a socket, or Kafka. It parses the message, processes it, and raises alerts if required. Everything is done in streams to allow for parallelism.

Creation of the parser uses `ParserFactory.java`, which can parse according to `packetformat.json` or some user-specified custom parser.

GlobalSFAProcessor.java: This class is the runtime DSFA processor. It takes events as input and outputs alerts. Contained in this processor is functionality for reordering events based on their timestamp, tracking stateful variables across events, and advancing all possible instantiations of the DSFA. Critical to the function of the DSFA is an expression tree of binary/boolean operators that assist in evaluating the predicates attached to each transition in the DSFA. See the `expressions` sub-directory for details.

A.1.3 Local Verifier

The local verifier has three dependencies: `cppkafka`, `rapidjson` and `antlr`. The primary files are:

main.cpp: Like `GenerateSFA.java` of the global verifier, the local C++ version is responsible for constructing the state machine from the provided files and processing input events coming from either a file or a socket. The overall flow of the local verifier mirrors that of the global verifier, except that this one is implemented in C++ with none of the Flink support for automatic scaling and fault tolerance: after receiving an event, the event is parsed using the `PacketParser` class and sent to the local SFA processor (described below). The key difference is that the objective of this version is to decide whether the event should be suppressed and output it if not. Events are only suppressed if all state machines agree that they are suppressible.

SFAProcessor.cpp: This is the local, C++ version of `GlobalSFAProcessor.java`. Like other portions of *Aragog*'s local components, the local SFA processor implements a stripped-down, slightly modified version of the global verifier's functionality. In this case, the local node is tracking its view of the global state of the system, given only the locally observed events. As such, it does not need to worry about event reordering or location-variable tracking, which simplifies the implementation and leads to improved performance.

A.2 Firewall Demo

We include in the repository an example experiment involving firewalls and verifiers that emulates a portion of the experimental methodology of Section 9. This experiment expects the user to have a small cluster of machines that can play the

role of each type of node. CloudLab is one viable option and we include configurations to assist in allocating such a cluster. The included code configures the topology of Figure 19.

The setup file, `Setup/setup.sh`, installs the required software on each machine in the user's cluster and also installs IP route rules that create an overlay corresponding to the topology referenced above.

Overall, the experiment consists of four external nodes, four internal nodes on a single LAN, and four firewall nodes interposing between the two groups. The firewalls are configured as two high-availability groups with one primary and one hot standby per group. Each primary-standby group shares a virtual IP with the VRRP protocol. We base the traffic between external nodes and internal nodes on traffic models from DCTCP [9].

The rules that are installed in the firewall are simple. Internal nodes can communicate with each other and initiate connections to external nodes. External nodes cannot initiate connections to internal nodes.

Alongside the firewall, each firewall node also runs the verifier, which computes filters and suppression. A single global verifier node runs both the Apache Kafka and Apache Flink deployments. Kafka is responsible for receiving and pipelining the events from all of the local verifiers. Flink is responsible for executing the global verifier.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution

Yigong Hu
Johns Hopkins University

Gongqi Huang
Johns Hopkins University

Peng Huang
Johns Hopkins University

Abstract

Misconfiguration is a major cause of system failures. Prior solutions focus on detecting *invalid* settings that are introduced by user mistakes. But another type of misconfiguration that continues to haunt production services is *specious configuration*—settings that are valid but lead to unexpectedly poor performance in production. Such misconfigurations are subtle, so even careful administrators may fail to foresee them.

We propose a tool called Violet to detect specious configuration. We realize the crux of specious configuration is that it causes some slow code path to be executed, but the bad performance effect cannot always be triggered. Violet thus takes a novel approach that uses selective symbolic execution to systematically *reason about* the performance effect of configuration parameters, their combination effect, and the relationship with input. Violet outputs a performance impact model for the automatic detection of poor configuration settings. We applied Violet on four large systems. To evaluate the effectiveness of Violet, we collect 17 *real-world* specious configuration cases. Violet detects 15 of them. Violet also identifies 11 unknown specious configurations.

1 Introduction

Software is increasingly customizable. A mature program typically exposes hundreds of parameters for users to control scheduling, caching, *etc*. With such high customizability, it is difficult to properly configure a system today, even for trained administrators. Indeed, numerous studies and real-world failures have repeatedly shown that misconfiguration is a major cause of production system failures [32, 43, 45, 60].

The severity of the misconfiguration problem has motivated solutions to detect [35, 61, 63], test [37, 57], diagnose [19, 21, 50, 52, 54] and fix [39, 48, 53] misconfiguration. While these efforts help reduce misconfiguration, the problem remains vexing [1–3, 5–10, 17, 18, 31]. They focus on catching *invalid* settings introduced due to user mistakes. But another type of misconfiguration that haunts production systems, yet not well addressed, is valid but poor configuration. For simplicity, we call them **specious configuration**.

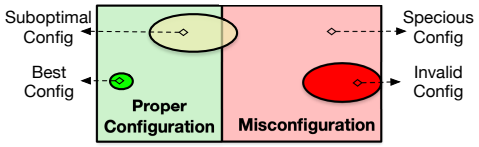


Figure 1: Value space of a configuration

Specious configuration has a broad scope. In this paper, we focus on—and use the term to refer to—valid settings that lead to extremely poor *performance*, which is a common manifestation in production incidents. This scope of focus is different from suboptimal configuration (Figure 1). The latter happens when a setting does not yield the best performance, but the performance is still acceptable. This scope is also complementary to efforts on automated configuration performance tuning [33, 51, 62, 64] to search for the best setting.

Take a real-world specious configuration that caused a service outage as an example. An engineer changed the request tracing code from a hard-coded policy (always tracing) to be configurable with a tracing `rate` parameter. This `rate` parameter was initially set to 0.0. To retain the same tracing behavior as before, she decided to change the parameter to 1.0. Based on her understanding, this change will turn on the tracing for all *message* requests that come from *internal* users. But unfortunately, there was a subtle caveat in the code that caused the actual effect to be turning on tracing for *all* requests from *all users*, which quickly overloaded all web servers as well as the backend databases, leading to a catastrophic service outage. Interestingly, before rolling out this specious configuration to production, the change in fact went through a canary phase on a small-scale testing cluster, which unfortunately did not manifest dramatic failure symptoms.

Empirical evidence suggests that specious configuration like the above is prevalent. Yin et al. [60] shows that misconfiguration in the form of legal parameters has similar or higher percentage than illegal parameters. Facebook reports [49] that more than half of the misconfiguration in their high-impact incidents during a three-month period are subtle, "valid" config-

urations. A recent study [51] on performance configurations in distributed systems reports a similar finding.

To reduce specious-configuration-induced incidents, we need to proactively detect it before production. However, what makes specious configuration subtle to detect is that its value is not a unconditionally poor choice. Rather, the setting is only problematic under certain combination with some other parameters, input, and/or environment. Currently, administrators either informally estimate the impact based on their experience, or experimentally measure it by black-box testing the program with configuration. However, neither of the approaches is sufficient to reliably capture the pitfalls.

Through analyzing real-world cases (Section 2), we realize that the crux of specious configuration lies in the fact that some slow code path in the program or library gets executed; but this effect can be only triggered with certain input, other configurations, and environment. Therefore, we argue that analytical approaches are needed to *reason about* the configuration settings' performance implications under a variety of conditions. We propose a novel analytical tool called VIOLET that uses symbolic execution [24, 38] to analyze the performance effect of configuration at the code level.

The basic idea of Violet is to systematically explore the system code paths with symbolic configuration and input, identify the constraints that decide whether a path gets executed or not, and analytically compare different execution paths that are explored. Violet derives a configuration performance impact model as its analysis output. A Violet checker leverages this model to contiguously catch specious configuration in the field. Making this basic idea work for large system software faces several challenges, including the intricate dependency among different parameters, the efficiency of symbolic execution for performance analysis, complex input structure, and path explosion problems. Violet leverages program analysis and selective symbolic execution [26] to address these challenges.

We implement a prototype of the Violet toolchain, with its core tracer built as plugins on the S^2E platform [26], the static analyzer built on LLVM [40], and the trace analyzer and checker built as standalone tools. We successfully apply Violet on four large systems, MySQL, PostgreSQL, Apache and Squid. Violet derives performance impact models for 471 parameters. To evaluate the effectiveness of Violet, we collect 17 real-world specious configuration cases. Violet detects 15 cases. In addition, Violet exposes 11 unknown specious configuration, 8 of which are confirmed by developers.

In summary, this paper makes the following contributions:

- An analytical approach to detect specious configuration using symbolic execution and program analysis.
- Design and implementation of an end-to-end toolchain Violet, and scaling it to work on large system software.
- Evaluation of Violet on real-world specious configuration.

The source code of Violet is publicly available at:

https://github.com/OrderLab/violet

2 Background and Motivation

In this Section, we show a few cases of real-world specious configuration from MySQL to motivate the problem and make the discussion concrete. We analyze how specious configuration affects system performance at the *source code* level. We choose MySQL because it is representative as a large system with numerous (more than 300) parameters, many of which can be misconfigured by users and lead to bad performance.

2.1 Definition

A program expects its configuration parameters to obey certain rules, *e.g.*, the path exists, the min heap size does not exceed the max size. Invalid configurations violate those rules and usually trigger assertions or errors.

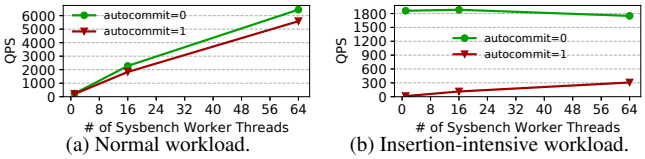
We define specious configuration to be settings that are valid but cause the software to experience bad performance when deployed to production. Admittedly, bad performance is a qualitative criterion. Like prior work, we focus on those issues that cause severe degradation and hurt usability. Ultimately, only users can judge whether the performance slowdown is sub-optimal but tolerable or it is intolerable.

Specious configuration has two classes. One is purely about performance, *e.g.*, buffer size, number of threads. Another class is settings that change the software functionality but the changes also have performance impact. Both classes are important and occur in real-world systems. For the latter class, users might want the enabled functionality and are willing to pay for the performance cost. Thus, whether the setting is specious or not depends on users' preferences. Our solution addresses both forms. Its focus is to analyze and explain the quantitative performance impact of different settings, so that users can make better functionality-performance trade-offs.

2.2 Case Studies

`autocommit` parameter controls the transaction commit behavior in MySQL. If `autocommit` is enabled, each SQL statement forms a single transaction, so MySQL will automatically perform a commit. If `autocommit` is disabled, transactions need to be explicitly committed with `COMMIT` statements. While `autocommit` offers convenience (no explicit commit required) and durability benefits, it also has a performance penalty since every single query will be run in a transaction. For some users, this performance implication may not be immediately apparent (especially since it is enabled by default). Even if users are aware of the performance trade-off, they might not know the degree of performance loss, only to realize the degradation is too much after deploying it to production. Indeed, there have been user-reported issues due to this setting [13, 15, 60], and the recommended fix is to disable `autocommit`, and manually batch and commit multiple queries in one transaction.

To quantify the performance impact, we use sysbench [16] to measure MySQL throughput with `autocommit` configura-



(a) Normal workload. (b) Insertion-intensive workload.

Figure 2: MySQL throughput for `autocommit` under two workloads.

```
1  int write_row() {
2   if (autocommit) {
3     ...
4     trx_commit_complete();
5   } else {
6     trx_mark_sql_stat_end();
7   }
8  }
9
10
```

```
ulint trx_commit_complete() {
  if (flush_at_trx_commit==1) {
    log_group_write_buf();
    fil_flush();  costly operation
  } else if (flush_at_trx_commit==2) {
    log_group_write_buf();
  } else {
    /* do nothing */
  }
}
```

Figure 3: Simplified code snippet from MySQL related to `autocommit`. The elements with orange-colored background represent configuration variables, and the pink ones represent slow operations.

tion set to be `ON` and `OFF`. The size of the database is 10 tables and 10K records per table. We run both a normal workload that consists of 70% read, 20% write and 10% other operations, and an insert-intensive workload. Figure 2 shows the result. We can see that in the normal workload (Figure 2a), the performance difference between `ON` and `OFF` are small. But in insertion-intensive workload (Figure 2b), enabling `autocommit` causes dramatically worse (6×) performance.

Figure 3 shows the code relevant to `autocommit`. We can see that the `autocommit` setting determines whether function `trx_commit_complete()` will be invoked. In this function, another parameter `flush_at_trx_commit`[1] further determines which path gets executed. When that parameter is set to 1, compared to 2, an additional `fil_flush` operation will be incurred, which has a complex logic but essentially will flush the table writes cached by the OS to disk through the `fsync` system call. The cost of `fsync` is the major contributor to the bad performance of `autocommit` mode; if `flush_at_trx_commit` is 2 or 0, the performance impact of `autocommit` mode will be much smaller. In addition, the function in which `autocommit` is used—`write_row()`—is called when handling `write` type queries but not `select` type queries. Therefore, the performance hit only affects insertion/update-intensive workloads.

`query_cache_wlock_invalidate` controls the validation of the query cache in MySQL. Normally, when one client acquires a `WRITE` lock on a MyISAM table, other clients are *not* blocked from issuing statements that read from the table if the query results are present in the query cache. The effect of setting this parameter to 1 is that upon acquisition of a `WRITE` lock for a table, MySQL invalidates the query cache that refers to the locked table, which has a performance implication.

As Figure 4 shows, enabling this parameter leads to the `free_query` operation (❶). Different from the `autocommit` case, this operation itself is not costly. But for other clients that attempt to access the table, they cannot use the associated

[1] Its full name in MySQL is `innodb_flush_log_at_trx_commit`. We abbreviate it and some other parameter names in this paper for readability.

```
void mysql_parse(THD *thd) {
  if (send_result_to_client(thd) <= 0) {
    mysql_execute_command(thd);  ❷
  }
}
int mysql_execute_command(THD *thd) {
  case SQLCOM_SELECT:
❸   open_and_lock_tables(thd, all_tables);
    break;
  case SQLCOM_LOCK_TABLES:
    lock_tables_open_and_lock_tables(thd);
    if (query_cache_wlock_invalidate)  ❶
      invalidate_query_block_list();
}
void invalidate_query_block_list() {
  free_query(list_root->block());
}
```

incoming queries not in query cache

free query cache

Figure 4: Code affected by `query_cache_wlock_invalidate`.

```
uint64_t log_reserve_and_open(uint len) {
  if (len >= log->buf_size / 2) {
    log_buffer_extend((len + 1) * 2);
  }
  len_upper_limit = LOG_BUF_WRITE_MARGIN + (5 * len) / 4;
  if (log->buf_free + len_upper_limit > log->buf_size) {
    mutex_exit(&(log->mutex));
    log_buffer_flush_to_disk();
    goto loop;
  }
}
```

Figure 5: Code affected by `innodb_log_buffer_size`.

query cache (❷), forcing them to open the table and wait (❸) while the write lock is held. Therefore, the effect is additional synchronization that decreases the system concurrency, which in turn can severely hurt the overall system query throughput.

Similar to `autocommit`, the performance effect depends on the parameters, execution environment and workloads. Specifically, the bad performance is only manifestable with the combination of MyISAM tables, `LOCK TABLES` statements and other clients doing `select` type queries on the locked table.

`innodb_log_buffer_size` determines the size of the buffer for uncommitted transactions. The default value (8M) is usually fine. However if MySQL has transactions with large blob/text fields, the buffer can fill up very quickly and incur performance hit. As shown in Figure 5, the parameter setting has two possible performance impacts: (1) if the length of a new log is larger than half of the `buf_size`, the system will extend the buffer first by calling `log_buffer_extend`, which in normal cases mainly involves memory allocation. But if other threads are also extending the buffer, additional synchronization overhead is incurred. If the buffer has pending writes, they will be flushed to disk first; (2) if the `buf_size` is smaller than the free size plus the length of new log, MySQL will trigger a costly synchronous buffer flush operation.

2.3 Code Patterns

Based on the above and other cases we analyze, we summarize four common patterns on how a specious configuration affects the performance of a system at the source code level:

1. The parameter causes some expensive operation like the `fsync` system call to be executed.
2. The parameter incurs additional synchronization that itself is not expensive but decreases system concurrency.

3. The parameter directs the execution flow towards a slow path, *e.g.*, not using cached result.
4. The parameter triggers frequent crossings of some threshold that leads to costly operations.

The general characteristic among them is that specious configuration controls a system's execution flows—different values cause the program or its libraries to execute different code paths. However, the performance impact is also *context-dependent*—a specious configuration is bad only when its value and other relevant factors together direct the system to execute a path that is significantly slower than others.

2.4 Approaches to Detect Specious Config

To detect specious configuration, operators rely on experience or manuals, which are neither reliable nor comprehensive. A more rigorous practice is to test the system together with configuration and quantitatively measure the end-to-end performance like throughput. However, if the testing does not have appropriate input or related parameters, the performance issue will not be discovered. Also, because the testing is carried out in a black-box fashion, the approach is *experimental*. The results are tied to the testing environment, which may not have the same hardware, dependencies or scale as the production. For example, in the incident described in Section 1, that specious configuration was tested, and the result showed a slight increase of logging traffic to a dependent database. But this increase was deemed small, so it passed the testing.

We argue that while the experimental approach is indispensable, it alone is insufficient to catch specious configuration. We advocate developing *analytical* approaches for *reasoning* about configurations' performance effect from the system code. The outcome from an analytical approach includes not only a conclusion, but also answers to questions "how the parameter affects what operations get executed?", "what kind of input will perform poorly/fine?", "does the effect depend on other parameters?", *etc.* In addition, the analysis should enable *extrapolation* to different contexts, so users can project the outcome with respect to specific workload or environment.

A potential approach is static analysis. Indeed, we can leverage the code patterns in Section 2.3 to detect potential specious configuration. However, mapping them at concrete code construct level requires substantial domain knowledge. Also, the performance effect involves many complex factors that are difficult to be deduced by pure static analysis.

The observations in Section 2.3 lead us to realize that the crux is some slow path being conditionally executed. Thus, we can transform the problem of detecting specious configuration to the problem of finding slow execution flow plus deducing the triggering conditions of the slow execution.

3 Overview of Violet

We propose an analytical approach for detecting specious configuration, and design a tool called VIOLET. Violet aims to comprehensively reason about the performance effect of system configurations: (1) explore the system without being limited by particular input; (2) analyze the performance effect without being too tied to the execution environment.

Our insight is that the subtle performance effect of a specious parameter is usually reflected in different *code paths* getting executed, depending on *conditions* involving the parameter, input and other parameters, and these paths have significant *relative* performance differences. Based on this insight, Violet uses symbolic execution with assistance of static analysis to thoroughly explore the influence of configuration parameters on program execution paths, identify the conditions leading to each execution, and compare the performance costs along different paths. After these analyses, Violet derives a configuration performance impact model that describes the relationship between the performance effect and related conditions. In this Section, we give an overview of Violet (Figure 6). We describe the design of Violet in Section 4.

3.1 Symbolic Execution to Analyze Performance Effect of Configurations

Background. Symbolic execution [24, 38] is a popular technique that systematically explores a program. Different from testing that exercises a single path of the program with concrete input, symbolic execution explores multiple paths of the program with symbolic input and memorizes the *path constraints* during its exploration. When a path of interest (e.g., with `abort()`) is encountered, the execution engine generates an input that satisfies the constraint, which can be used as a test case. Compared to random testing, symbolic execution systematically explores possible program paths while avoiding redundancy. Consider this snippet:

```
void foo(int n) { if (n > 1000) bar(n); else bazz(n); }
```

Testing may blindly test the program many times with different `n`, e.g., 1, 10, 20, etc., but they all exercise the same path without triggering the call to `bar()`. If we use symbolic execution, we can explore the two paths of `foo` by deriving only two concrete values of `n` to satisfy the path constraints.

Basic Idea. Configuration is essentially one type of input to a program. The basic idea of Violet is simple—make the parameters symbolic, measure the cost along each execution path explored, and comparatively analyze the costs. The path constraints that the symbolic execution engine memorizes characterize the conditions about whether and when a parameter setting is potentially poor. Take Figure 3 as an example. Violet makes variable `autocommit` symbolic. Function `write_row` will fork at line 2. The first path goes into the `if` branch, with a constraint `autocommit == 1`. When `trx_commit_complete` is called in the first path, it encounters another parameter `flush_at_trx_commit`, which is also made symbolic. Two additional paths are forked within that function. While exploring these paths, Violet records a set of performance cost metrics.

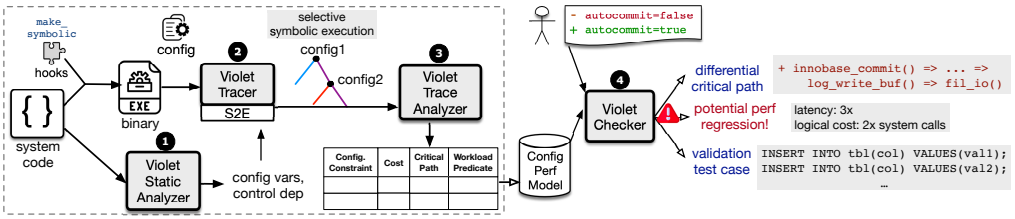


Figure 6: Overview of Violet.

Since the subtle performance effect of specious configuration is often only triggered under specific input, besides configuration parameters, Violet can also make the input symbolic. For the example in Figure 3, the input will determine whether the `write_row` function will be called or not. Only `insert` type queries will invoke `write_row`. This input constraint will be recorded so the analysis later can identify what *class* of input can trigger the specious configuration.

3.2 Violet Workflow

Figure 6 shows the workflow of Violet. The input to Violet is system code and target configuration. We require source code to identify the program variables corresponding to parameters. In addition, as we discuss later (Section 4.3), Violet uses static analysis to assist the discovery of dependent parameters. To symbolically execute the target system, we leverage a state-of-the-art symbolic execution platform S^2E [26] and insert hooks into the system code to make parameters and input symbolic. We design the Violet execution tracer as S^2E plugins to record the performance results to a trace during state exploration. The Violet trace analyzer conducts comparative cost analysis, differential critical path analysis, *etc.* The output is a configuration performance impact model that describes the relationship among configuration constraints, cost, critical path, and input predicate.

Violet further provides a checker to deploy with the software at user sites. The checker consumes the constructed configuration impact model to continuously detect whether a user-site configuration file or update can potentially lead to poor performance. Upon the detection of potential specious configuration, the Violet checker reports not only the absolute performance result, but also the logical cost and critical path to explain the danger. The checker also outputs a validation test case based on the input predict that provides hints to users about what input can expose the potential performance issue.

4 The Design of Violet

In this Section, we describe the Violet design (Figure 6). We need to address several design challenges. First, configurations have intricate dependencies among themselves and with the input, but making all of them symbolic easily leads to state space explosion. Second, conducting performance analysis in symbolic execution is demanding due to lack of explicit assertion point, mixed costs, overhead, etc. Third, deriving performance model from code requires balance between being generalizable (not too tailored to specific input or environment) and being realistic (reflects costs in real executions).

4.1 Make Config Variable Symbolic

The starting point for Violet is to make parameters symbolic. A naïve way is to make the entire configuration file a symbolic blob. While this approach is transparent to the target program, it easily leads to path explosion even at the program initialization stage. An improvement could be only making the configuration value string symbolic during parsing. e.g., `make_symbolic(value, 2); buf_size=atoi(value);` But the execution would still spend significant time in the parsing (`atoi`). Also the parameter value range will be limited by the string size, e.g., only explore `buf_size` from 0 to 99.

We should identify the program variables that store configuration parameters and directly make these variables symbolic. Prior works [56,57] observe that the mature software typically uses uniform interfaces such as an array of `struct` to store parameters. Thus they annotate these interfaces to extract variable mappings in static analysis. For Violet, we need to additionally identify the parameter type and value constraints defined by the program (e.g., 1 to 10) to restrict the symbolic value. This is because we are only interested in exploring the performance effect of *valid* values.

Since typically all the config variables are readily accessible after some point during initialization, we take a simple but accurate approach: insert a hook function directly in the source code right after the parsing function and programmatically enumerates these variables and make them symbolic using their type and other info. In this hook function, we read an external environment variable `VIO_SYM_CONFIGS` to decide which target parameter(s) to make symbolic.

Take MySQL as an example. Its configuration parameters are represented by a number of `Sys_var_*` data structures in the code, depending on the parameter's type. We add a `make_symbolic` API to these data structures, which uses the type, name, value range information to call the Violet library to make the backing store symbolic. Figure 7 shows an example of the added hook API. Then after MySQL finishes parsing its configurations, we iterate through all configuration variables (Figure 8), which are stored in a global linked list called `all_sys_vars`. If the parameter is in the target set, we invoke its new `make_symbolic` API.

```
template <typename T>
class Sys_var_unsigned: public sys_var {
public:
  Sys_var_unsigned(const char *name, T min_val, T max_val, ...) {
    option.min_value= min_val;
    option.max_value= max_val;
    ...
  }
  bool global_update(THD *thd, set_var *var) {
    global_var(T)= var->save_result.ulonglong_value;
    return false;
  }
  ...
+ bool make_symbolic() {
+   violet_make_symbolic(global_var_ptr(), sizeof(T), option.name);
+   violet_assume((unsigned)(*global_var_ptr()) <= option.max_value);
+   violet_assume((unsigned)(*global_var_ptr()) >= option.min_value);
+   return true;
+ }
}
```

Figure 7: Add API to one config. data structure in MySQL.

```
static int get_options(int *argc_ptr, char ***argv_ptr)
{
  my_init_dynamic_array(&all_options, sizeof(my_option));
  for (opt= my_long_options; opt < my_options_end; opt++) {
    insert_dynamic(&all_options, (uchar*) opt);
    ...
  }
+ violet_parse_config_targets();
+ violet_make_mysql_options_symbolic();
  return 0;
}

+ void violet_make_mysql_options_symbolic()
+ {
+  for (sys_var *var=all_sys_vars.first; var; var= var->next)
+    if (is_config_in_targets(var->name.str))
+      var->make_symbolic();
+ }
```

Figure 8: Call symbolic hooks after config. parsing in MySQL.

4.2 Make Related Config Symbolic

The performance effect of a parameter usually depends on the values of other parameters. Thus, if we only make one parameter symbolic while leaving other parameters concrete, we will only explore incomplete execution paths and potentially miss some problematic combination that leads to bad performance. A straightforward solution is to make all parameters symbolic. Since symbolic execution only forks if a symbolic value is used branch conditions, this approach seems to be feasible. However, the problem with this approach is that most combinations of configuration parameters are unrelated but will be explored during symbolic execution.

Figure 9 illustrates the problem. Suppose we are interested in the performance effect of `opty`. If we simply make all parameters (`optx`, `opty`, `optz`) symbolic in hope of exploring the combination effect, there will be at least 6 execution paths being explored. But `opty` is unrelated to `optx` and `optz`. The performance impact of `opty` is only determined by the cost of its branches. For large programs, the target parameter could be used deep in the code. Including unrelated parameters in the symbolic set can cause the symbolic execution to waste significant time or get stuck before reaching the interesting code place to explore the target parameter. The analysis result can also cause confusions. For example, it might suggest only when `optx>100 && optz==FILE && opty` is true will there be a performance issue and miss detecting specious configuration when `opty` is true but `optx <= 100` or `optz != FILE`.

Therefore, instead of making all parameters symbolic, we carefully choose the set of parameters to symbolically execute

```
void main() {
  if (optx > 100)
    init_x();
  ...
  if (opty)
    task1();
  else
    task2();
}
void init_x() {
  if (optz == FILE)
    create_file();
}
```

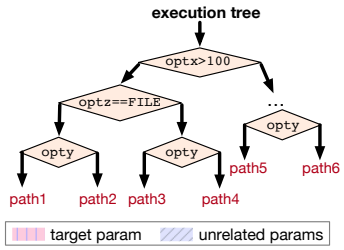


Figure 9: Making unrelated parameters symbolic results in excessive state explorations and confusing conclusions.

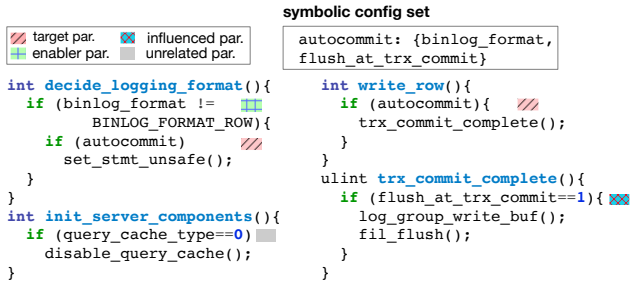


Figure 10: Symbolic config set based on control dependencies.

4.3 Discover Control Dependent Configs

Violet statically analyzes the control dependency relationship of parameters to determine a reduced symbolic *parameter set*. The static analysis result can significantly help mitigate the path exploration problem during symbolic execution phase.

For a target parameter C, Violet identifies two kinds of related parameters to put in its symbolic set. The *enabler parameters* are those that C is control dependent on. The *influenced parameters* are those that are control dependent on C. Figure 10 shows an example. For target parameter `autocommit`, it is used in `decide_logging_format` and `write_row`, it has an enabler parameter `binlog_format`, which decides if `autocommit` will be activated. `autocommit` itself influences the performance effect of parameter `flush_at_trx_commit`. Thus, for `autocommit`, the set of related parameters to make symbolic together is {`binlog_format`, `flush_at_trx_commit`}.

Informally, program element Y is control dependent on element X if whether Y's executed depends on a test at X. More formally, control dependency is captured by postdominator relationship in program Control Flow Graph (CFG). Node b in the CFG postdominates node a if every path from a to the exit node contains b. Y is control dependent on X if there is a path $X \rightarrow Z_1 \rightarrow \ldots \rightarrow Z_n \rightarrow Y$ such that Y postdominates all Z_i and Y does *not* postdominate X. We use postdominator as a building block for our analysis. But our notion of control dependency is broader than the classic definition. For example, `if (X) { if (Z1) { if (Z2) { if (Y) { foo(); } } } }` , the classic definition does *not* regard `X` and `Y` as being control-dependent, because `Y` does *not* postdominate `Z1` or `Z2`;

it regards `Z2` and `Y` as being control-dependent. But for us, all the four parameters are control dependent.

Our analysis is divided into two steps. The first step computes the enabler parameters. Violet builds a call graph of the program. For target parameter p, it locates the usage points of p and extracts the call chains starting from the entry function to the function f that encloses a usage point. If any caller g in the call chain uses some other parameter q, we check if the callsite in g that eventually reaches f is control dependent on the usage point of parameter q in g. If so, q is added to the enabler parameter set of p. Violet identifies enabler parameters within f through intra-procedural control dependency. Our technical report [34] lists the algorithm.

In the second step, Violet calculates the influenced parameters from the computed enabler parameter sets of all parameters. The related config set is a union of the influenced set and enabler set. We also capture control dependency that involves simple data flow. For example,

```
void query_cache_init() {
  if (query_cache_type == 0)
    m_cache_is_disabled = TRUE;
}

bool is_disabled() {
  return m_cache_is_disabled;
}
```

any parameter that is control dependent on the regular variable `m_cache_is_disabled` or return value of `is_disabled()` is also considered to be related to parameter `query_cache_type`.

The static analysis result can be inaccurate due to imprecision in the alias analysis, call graph, infeasible path problem, *etc.* Our general principle is to be conservative and over-approximate the set of related parameters for a target parameter. During symbolic execution, having a few false control dependent parameters does not greatly affect the performance or analysis conclusion and they can manifest through the symbolic execution log if they do cause issues.

4.4 Execute Software Symbolically

After the target software is instrumented with the symbolic execution hooks, Violet symbolically executes the software with a concrete configuration file. The hook function reads the `VIO_SYM_CONFIGS` environment variable and makes symbolic the program variables corresponding to the specified parameter. In addition, the function parses the control dependency analysis (Section 4.3) result file and makes variables in the related parameter set symbolic as well. Other parameters' program variables get the concrete values from the configuration file. Besides parameters, Violet can also make program input symbolic to explore its influence on the configuration's performance impact. This is done through either symbolic arguments (`sym-args`) or identifying the input program variables and inserting `make_symbolic` calls in the code.

4.5 Profile Execution Paths

To measure the symbolic parameters' performance effect, Violet implements a tracer on top of the symbolic execution engine, specifically as a set of plugins on the S^{2}E platform.

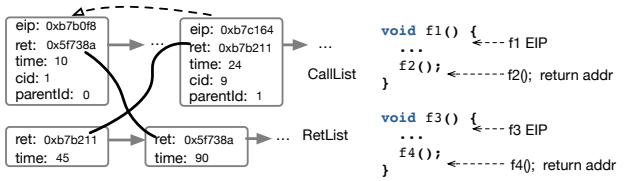


Figure 11: Match call/return records.

Measure Function Call Latency. We measure function call latency by capturing the call and return signals emitted by S^{2}E during symbolic execution. To calculate the latency, a straightforward way is to maintain a stack of call record and pops the top element upon receiving a return signal. This algorithm assumes that the call/return signals are paired and the callee's return signal comes before the caller's. But we observe this assumption does not always hold under S^{2}E. We use a safer method based on return addresses to calculate latency. In particular, the Violet tracer records the `EIP` register value, return address, and timestamp on each call and return signal. The records are stored in two lists. Later, the tracer matches call record list with return record list based on return address fields (Figure 11). The latency for a matched function call is the return record's timestamp minus the call record's timestamp. The total latency of each state (execution path) can be obtained from the latency of the root function call.

For multi-threaded programs, function calls from different threads can get mixed up. To address this issue, the Violet tracer stores the current thread id in each profile record and partitions the call and return lists by thread id.

Re-Construct Call Paths. The tracer records the function call profile to break down total latency and to enable differential critical path analysis (§4.6). To get the call chain relationship, instead of costly stack frame walk, the tracer uses a simple method with little overhead that just assigns each call record a unique incrementing `cid`. Later, the tracer reconstructs the call chain by iterating through all call records in order. If (1) call record A's `cid` is larger than call record B's `cid`, (2) the return address of A is larger than B's `EIP` (the start address of that function), and (3) the difference of the two addresses is smallest among all other pairs (i.e., B's start address is closest to the return address in A), then we assign A's `parentId` to be B's `cid` and update the current distance.

Measure Logical Costs. Besides absolute latency, we also measure a set of logical cost metrics by a similar method of capturing low-level signals from S^{2}E. In particular, for each execution path, we measure the number of instructions, the number of system calls, the number of file I/O calls, the amount of I/O traffic, the number of synchronization operations, network calls, etc. These logical costs are useful to surface performance issues other than just long latency. They are also crucial for reducing the test environment's biases and enabling extrapolation of the result to different settings. For example, if the tracer finds one execution path has a much higher number of `write` syscalls compared to other paths whereas

Configuration Constraint	Cost	Workload Predicate
`autocommit!=0 && flush_log_at_trx_commit==1`	2.6 s, {`log_write_buf→fil_flush`}, 17K syscalls, 100 I/O insts, ...	`sql_command==INSERT`
`autocommit!=0 && flush_log_at_trx_commit==2`	1.7 s, {`log_write_buf`}, 16.9K syscalls	`sql_command==INSERT`
`autocommit!=0 && flush_log_at_trx_commit!=1 && flush_log_at_trx_commit!=2`	1.2 s, {}, 16.9K syscall	`sql_command==INSERT`
`autocommit==0`	0.6 s, {`trx_mark_sql_stat_end`}, 16.8K syscalls	`sql_command==SELECT||...`

Table 1: Example raw cost table Violet generates for `autocommit` parameter from symbolic execution of MySQL code in Figure 3.

their latencies are similar. This could be an artifact of the test server having a powerful hard disk or a large buffer cache. But the software might perform poorly in a different environment. The Violet tracer maintains a separate performance profile for each execution path (state) so we can compare the performance effect of different paths. We also need to record the path constraints to identify the parameter combination and the *class* of input that leads to the execution path. The tracer records the final path constraint when an execution path terminates or it exceeds some user-specified cost threshold.

4.6 Analyze State Traces

Once the symbolic execution finishes, the Violet trace analyzer parses the performance traces. It then builds a cost table. Each row represents a state (path) that was explored in symbolic execution. The analyzer does a pair-wise comparison of performance in different rows. If the performance difference ratio exceeds a threshold (default 100%), the analyzer marks that state suspicious. The analyzer compares not only the absolute latency metric but also the collected logical metrics. Even if the latency difference does not exceed the threshold but some logical metric does, the analyzer still marks the state.

Not all pair comparisons are equally meaningful when the symbolic execution explored multiple symbolic variables. To elaborate, assume our target parameter is `autocommit`, which has a related parameter `flush_log`. Since both are made symbolic, one state could represent constraint `autocommit==0 && flush_log==1` and another state could represent constraint `autocommit==1 && flush_log==2`. In this case, comparing the costs of these two states is not very meaningful.

The analyzer tries to compare state pairs that are most "similar" first. Determining the similarity of two paths can be challenging. We use a simple approach: in one state's constraints formula, for each constraint involving a related parameter, if it also appears in the other state's formula, the similarity count is incremented by one. This method is imprecise as it merely checks the appearances, not constraint equivalence. For our use cases, the inaccuracies are generally acceptable. Besides, the analyzer can compare all pairs first, surface the bad state-pairs, and then we can decide the meaningfulness of the suspicious pairs.

For each pair that has a significant performance difference, the analyzer computes the *differential* critical path. It first finds the longest common subsequence of the call chain records in the two states. Then it creates a diff trace that stores the common records with performance metrics subtracted, as well as the records that only appear in the slower state. The analyzer finally locates the call record (excluding entry) with the largest differential cost and constructs the critical call path based on the `cid` and `parentId` of the call records.

When Violet makes the input symbolic, the path constraints in each state will contain constraints about the input. The analyzer separates the input related constraints as input predicate. This is useful to tell what *class* of input can expose the potential performance issue for the combination of parameter values that satisfies the configuration constraint in a state. The final output from the Violet analyzer is the configuration performance impact model that consists of the raw cost table (Table 1) with configuration constraints, cost metrics, and input predicate for each state, the state pairs that have significant performance difference, and the differential critical paths.

4.7 Continuous Specious Config Checker

Violet provides a standalone checker tool to detect specious configuration. It leverages the configuration performance impact model from the analyzer and validates a concrete user configuration file. The checker tool supports three modes:

1. Some config update introduces performance regression.
2. Some default parameter is poor for users' specific setup.
3. Code upgrade or workload change make old setting poor.

For scenario 1, the checker references the cost table and locates the state(s) that have configuration constraints satisfying the updated parameter' old value and the parameter's new value. If the state pair has significant performance difference, the checker alerts the operators and generates a test case based on the input predicate for operators to confirm the performance regression. For scenario 2, the checker validates if the state that the default value lies in appears in some poor state-pair. If so, it means this default value potentially performs significantly worse than another value. For scenario 3, if the system code changes, Violet rebuilds the cost impact table. The checker then identifies if some state in the new table performs much worse compared to the old cost table. If workload changes, the checker validates if cost table rows that previously satisfy the input predicate perform worse compared to rows that satisfy the input predicate now.

5 Scaling Violet to Large Software

In this section, we describe the challenges and our solutions for scaling Violet to large software.

5.1 Choice of Symbolic Execution Engine

We initially build Violet on the KLEE [24] symbolic execution engine because it is widely used and convenient to exper-

iment with. However, while KLEE works well on moderate-sized programs, it cannot handle large programs like MySQL. KLEE models the environment (POSIX runtime and libc) with simplified implementation. Large programs use many libc or system calls that are unimplemented or implemented partially/incorrectly, e.g., `fcntl`, `pread`, and socket. KLEE also does not support symbolic execution of multi-threaded programs. We spent several months patching KLEE to fix the environment model and add multi-threading support. When we were finally able to run MySQL with KLEE, it took 40 minutes to just pass initialization even without symbolic data.

We thus decided to switch to the S^{2}E platform [26]. S^{2}E uses real environment with complete OS and libraries. Executing large software would encounter almost no compatibility issues. In addition, S^{2}E uses QEMU and dynamic binary translation to execute a target program. For instructions that access symbolic data, they are interpreted by the embedded KLEE engine; but instructions that access concrete data are directly executed on host CPU. Overall, while the choice of using real environment in symbolic execution in general means slower analysis compared to using simplified models like KLEE, executing concrete instructions on host CPU offsets that slowness and allows S^{2}E to achieve significant speed-up. After migrating Violet to S^{2}E and with some minor adjustments, we can start MySQL server within one minute.

5.2 Handle Complex Input Structure

Since specious configuration is often only triggered by certain input, Violet makes input symbolic besides configuration. For small programs, the input type is typically simple, *e.g.*, an integer, a string, which is easy to be made symbolic. However, large programs' input can have very complex structure. If we make such complex input symbolic, the program may be stuck in the input parsing code for a long time and the majority of the input generated is invalid. For example, we make input variable `char *packet` (32 bytes) in MySQL symbolic and execute it in S^{2}E for 1 hour, which generates several hundred test cases, but none of which is a legal SQL query. Even after adding some additional constraints, the result is similar.

This challenge is not unique to our problem domain. Compiler testing [58] or fuzzing [11] also faces this challenge of how to generate valid input to programs like C compiler or DBMS. We address this problem through a similar practice by introducing workload templates. Instead of having the parser figure out a valid structure, we pre-define a set of input templates that have valid structures. Then we parameterize the templates so that they are not fixed, *e.g.*, the query type, insertion value, the number of queries, etc. In this way, we can make the workload template parameters symbolic.

5.3 Reduce Profiling Overhead

Profiling large programs can incur substantial overhead. We build Violet tracer using low-level signals emitted by S^{2}E rather than intrusive instrumentation. Nevertheless, symbolic execution is demanding for performance analysis as the program runs much slower compared to native execution. Fortunately, Violet cares about the relative performance between different paths. We can still identify specious configuration if the relative differences roughly match the native execution, which we find is true for most cases. Violet conducts differential analyses to capture performance anomalies. We describe three additional optimizations in Violet tracer.

First, the Violet tracer controls the start and end of its function profiler. This is because if we enable the function profiler at the very beginning, it can be overwhelmed by lots of irrelevant function calls. We add APIs in the tracer and will start the tracer when the target system finishes initialization and stop the tracer when the system enters the shutdown phase.

Second, the tracer avoids guest memory accesses and on-the-fly calculation. Accessing memory in an execution state goes through the emulated MMU in QEMU. Violet tracer only accesses and stores key information (most from registers) about the call/return signals. It defers the record matching, call chain and latency calculation to path termination.

Third, Violet will disable state switching during latency tracking if necessary. Since the function profiler calculates the execution time by subtracting the return signal timestamp from call signal timestamp, if S^{2}E switches to execute another state in between, the recorded latency will include the state switching cost. This in general does not cause serious problems because the costs occur in all states and roughly cancels out with our differential analysis. But in rare cases, the switching costs can distort the results. When this happens, Violet will force S^{2}E to disable state switching.

5.4 Path Explosion and Complex Constraints

A common problem with symbolic execution is path explosion, especially when the symbolic value is used in library or system calls. In addition, some library calls with symbolic data yield complex constraints that make the symbolic execution engine spend a long time in solving the constraints.

Violet leverages a core feature in S^{2}E, *selective symbolic execution* [26], to address this problem. Selective symbolic execution allows transition between concrete and symbolic execution when crossing some execution boundary, *e.g.*, a system call. Violet uses the Strictly-Consistent Unit-Level Execution consistency model, which silently concretizes the symbolic value before entering the boundary and adds the concretized constraint to the symbolic value after exiting the boundary. This consistency model sacrifices completeness but it would not invalidate the analysis result. To improve completeness, we add some relaxation rules in Violet without causing functionality errors: 1) if the library call does not add side effect, such as `strlen`/`strcmp`, we make the return value symbolic and remove the concretized constraint; 2) if the library call has side effect but does not hurt the functionality, such as `printf`, we directly remove the concretized constraint.

Software	Desc.	Arch.	Version	SLOC	Configs	Hook
MySQL	Database	Multi-thd	5.5.59	1.2M	330	197
Postgres	Database	Multi-proc	11.0	843K	294	165
Apache	Web server	Multi-proc-thd	2.4.38	199K	172	158
Squid	Proxy server	Multi-thd	4.1	178K	327	96

Table 2: Evaluated software. Hook: SLOC of core Violet hooks.

One issue we encounter with the S^{2}E silent concretization is that its `concretize` API will only concretize the symbolic variable. The symbolic variable can taint other variables (make them symbolic) when it is assigned to these variables, but these tainted variables are not concretized during silent concretization. Having these tainted variables remain symbolic can add substantial overhead. We thus add a new API in S^{2}E, `concretizeAll`, that concretizes not only the given symbolic variable but also its tainted variables. We implement this API by recording in each write operation a mapping from the symbolic expression to the target address in the memory object. Later when `concretizeAll` is called, we will look up the memory objects to find addresses that contain the same symbolic expression and also concretize them.

6 Implementation

We implement the major Violet components in C/C++. The Violet checker is implemented in Python. The Violet tracer is written as S^{2}E plugins and leverages S^{2}E's existing plugin to capture low-level signals. The Violet static analyzer is built on top of LLVM framework [40]. The Violet trace analyzer is implemented as a standalone tool.

In function profiling, for efficiency, the tracer captures the addresses instead of names of invoked functions. This means the analyzer needs to resolve the addresses to names. The problem is that the virtual address of the target program can change in each run. We address this issue by modifying the ELF loader of the S^{2}E Linux kernel to expose the `load_bias`. Then the tracer will record the offset from the `load_bias`. The analyzer can then use the offsets to resolve the names.

7 Evaluation

We evaluate Violet to answer several key questions:

- How effective is Violet in detecting specious configuration?
- Can Violet expose unknown specious configuration?
- How useful is Violet's checker to the user?
- What is the performance of Violet?

The experiments are conducted on servers with Dual Processor of Intel Xeon E5-2630 (2.20GHz, 10 cores), 64 GB memory, 1 TB HDD running a Ubuntu 16.04. Since S^{2}E engine runs in QEMU, we create a guest image of Debian 9.2.1 x86_64 with 4 GB memory for all the Violet tests.

7.1 Target Systems

We evaluate Violet on four popular and large (up to 1.2M SLOC) open-source software (Table 2): MySQL, PostgreSQL, Apache, and Squid. Violet can successfully analyze large multi-threaded programs (MySQL and Squid) as well as multi-process (PostgreSQL, Apache) programs.

The manual effort to use Violet on a target system is small, mainly required in two steps: (1) add configuration hooks (Section 4.1); (2) supply input templates (Section 5.2). The other steps in the workflow are automated.

Table 2 shows SLOC of the core hooks we add to the four systems. The hook size varies across systems. MySQL hooks are largest in size mainly because the system defines many (22) configuration types (`Sys_var_*`) so we need to add hook (about 7 SLOC) to each type. But the overall effort for different systems is small. The changes are typically contained in a few places with other codes untouched. In addition, most software rarely modifies the configuration data structure design, so the effort can carry through versions.

For (2), users typically already have some workload profiles. The effort needed is to parameterize and organize them into our format. In our experience with the four evaluated software, this process is straightforward and can be done in a few hours.

7.2 Detecting Known Specious Config

To evaluate the effectiveness of Violet we collect 17 *real-world* specious configuration cases from the four systems. Table 3 lists the case descriptions. We collect them from ServerFault [14], dba [4], blog posts [12], and prior work [19]. A case is marked as detected when Violet explores at least one poor state in its trace *and* the poor states enclose the problematic parameter value(s).

In total, Violet detects 15 of the 17 cases. Table 4 shows the detailed result. For each case, Table 4 lists the total states Violet explored, poor states, related configs, and maximum cost metric differences. The explored states include forks from related configurations and the symbolic workload parameters. In most cases, the specious configuration requires specific related settings to expose the issue. The high success rate of Violet comes from its in-vivo multi-path profiling, dependency analysis, and differential performance analysis.

Another aspect to interpret the high success rate is that the 17 cases we collect admittedly have a selection bias—all cases cause severe performance impact. This is reflected in the max diff column. If a misconfiguration only introduces mild performance issue, Violet may miss it due to the noises in symbolic execution. However, Violet's goal is to exactly target specious configuration that has severe performance impact, rather than suboptimal configurations.

Violet misses two Apache cases, c14 and c15. Triggering them requires enabling the HTTP KeepAlive feature in the workload. In our Apache workload templates, this feature is not part of the workload parameters and is disabled by default.

We describe two representative cases. MySQL c1 is the running example in the paper. Violet identifies four related parameters for `autocommit` and explores 88 states in total, 4 of which are identified as poor. The configuration constraints

Id.	Application	Configuration Name	Data Type	Description
c1	MySQL	`autocommit`	Boolean	Determine whether all changes take effect immediately
c2	MySQL	`query_cache_wlock_invalidate`	Boolean	Disable the query cache when after WRITE lock statement
c3	MySQL	`general_log`	Boolean	Enable MySQL general log query
c4	MySQL	`query_cache_type`	Enumeration	Method used for controlling the query cache type
c5	MySQL	`sync_binlog`	Integer	Controls how often the MySQL server synchronizes binary log to disk
c6	MySQL	`innodb_log_buffer_size`	Integer	Set the size of the buffer for transactions that have not been committed yet
c7	PostgreSQL	`wal_sync_method`	Enumeration	Method used for forcing WAL updates out to disk
c8	PostgreSQL	`archive_mode`	Enumeration	Force the server to swtich to a new WAL periodically and archive old WAL segments
c9	PostgreSQL	`max_wal_size`	Integer	Maximum number of log file segments between automatic WAL checkpoints
c10	PostgreSQL	`checkpoint_completion_target`	Float	Set a fraction of total time between checkpoints interval
c11	PostgreSQL	`bgwriter_lru_multiplier`	Float	Set estimate of the number of buffers for the next background writing
c12	Apache	`HostNamelookup`	Enumeration	Enables DNS lookups to log the host names of clients sending requests
c13	Apache	`Deny/Domain`	Enum/String	Restrict access to the server based on hostname, IP address, or env variables
c14	Apache	`MaxKeepAliveRequests`	Integer	Limits the number of requests allowed per connection
c15	Apache	`KeepAliveTimeOut`	Integer	Seconds Apache will wait for a subsequent request before closing the connection
c16	Squid	`cache`	String	Requests denied by this directive will not be stored in the cache
c17	Squid	`Buffered_logs`	Integer	Whether to write access_log records ASAP or accumulate them in larger chunks

Table 3: Description of 17 known specious configuration cases we collect in the four evaluated software.

Id.	Detect	Explored States	Poor States	Related Configs	Cost Metrics	Analysis Time	Max Diff*
c1	✓	88	17	4	Latency	6 m25 s	14.5×
c2	✓	24	3	1	Lat.&Sync.	3 m13 s	15.7×
c3	✓	224	88	5	I/O	19 m41 s	2.0×
c4	✓	787	100	2	Latency	53 m50 s	11.7×
c5	✓	494	44	3	Latency	17 m56 s	29.9×
c6	✓	891	12	5	I/O	112 m24 s	3.0×
c7	✓	89	7	2	Lat.&I/O	4 m6 s	4.3×
c8	✓	195	8	3	Latency	13 m8 s	1.8×
c9	✓	110	2	3	Lat.&I/O	15 m20 s	3.5×
c10	✓	231	13	7	Latency	23 m30 s	2.4×
c11	✓	61	9	2	Latency	13 m17 s	8.6×
c12	✓	34	4	2	Latency	7 m15 s	3.8×
c13	✓	50	5	3	Latency	6 m10 s	8.9×
c14	✗	112	0	2	Latency	3 m42 s	0.6×
c15	✗	23	0	3	Latency	6 m12 s	0.2×
c16	✓	81	1	0	Latency	433 m32 s	4.3×
c17	✓	3	1	0	I/O	1 m32 s	2.0×

Table 4: Violet detection result. Poor states are what Violet considers as suspicious. *: relative difference, $\alpha\times$ means $B = (1+\alpha) * A$.

of the four poor states describe the combination conditions for the 5 parameters to incur significant cost.

In c6, `innodb_log_buffer_size` controls the size of the log buffer. Interestingly, in this case, Violet determines the latency metric difference is not significant, but the I/O logical cost metric is. Specifically, Violet explores almost 100 different queries, and finds that in states with queries involving large row changes and a relatively small buffer size, the I/O metric—`pwrite` operations—is much larger than other states.

7.3 Comparison with Testing

We evaluate the 17 cases with testing as well. We use popular benchmark tools `sysbench` and `ab`. For each case, we set the target parameter and related parameters with concrete values from one of the poor states discovered. We enumerate the standard workloads in the benchmark to test the software with the configurations. Since the absolute performance result are difficult to judge, we use configurations from the good states and collect performance result with them as a baseline. If the performance difference ratio exceeds 100% (the same threshold used by Violet), we consider the case detected. In total, testing detects 10 cases, with a median time of 25 minutes.

Violet is not meant to replace configuration performance testing. In theory, exhaustive testing can expose all cases, but the cost of it is not affordable in practice. Violet systematically explores program states while avoids the redundancy in exhaustive testing (Section 3.1). Even though in some cases, as shown in Table 4, the Violet analysis time is relatively long, Violet is exploring the performance effects thoroughly, including the combination effect with other parameters and input. Therefore, the performance impact models Violet derives are complete. Once the exploration is done, the outcome can be reused many times while testing needs to be done repeatedly.

Another challenge with testing is to find the baseline for good performance. Our experiment above assumes the existence of good configuration, which users may not have. Violet, in comparison, conducts in-vivo, multi-path analysis, so it naturally has baselines to compare with. The analysis enables Violet to collect deeper logical metrics, which can reveal performance issues that end-to-end metrics may not find.

7.4 Exposing Unknown Specious Config

Besides detecting know specious configuration, we evaluate whether Violet can expose unknown specious configuration. We first apply Violet to derive performance models for all parameters if possible (Section 7.6). We then analyze the results for parameters not in the known case dataset (Section 7.2). We manually check (1) if some parameter's default or suggested value is in a poor state; (2) if a poor state of a parameter contains related parameters that are undocumented. The manual inspection involves checking the Violet output, the descriptions in the official documentation and tuning guide,

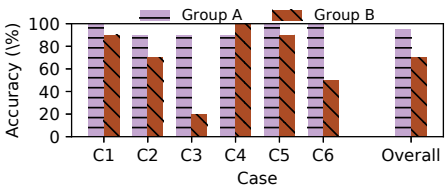


Figure 12: Overall accuracy of judgment in the user study.

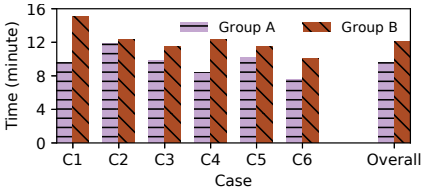


Figure 13: Average decision time in the user study.

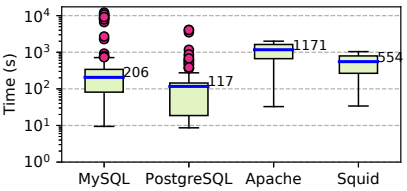


Figure 14: Violet analysis times for the configs in the four software.

Sys	Configuration	Performance Impact
Postgres	`vacuum_cost_delay`	Default value 20 ms is significantly worse than low values for write workload.
Postgres	`archive_timeout`	Small values cause performance penalties.
Postgres	`random_page_cost`	Values larger than 1.2 (default 4.0) cause bad perf on SSD for join queries.
Postgres	`log_statement`	Setting `mod` causes bad perf. for write workload when `synchronous_commit` is `off`.
Postgres	`parallel_setup_cost`	A higher value would avoid unnecessary parallelism when executing join query
Postgres	`parallel_leader_participation`	Enabling it can cause select join query to be slow if `random_page_cost` is high.
MySQL	`optimizer_search_depth`	Default value would cause bad performance for join queries
MySQL	`concurrent_insert`	Enable concurrent_insert would cause bad performance for read workload
Squid	`ipcache_size`	The default value is relatively small and may cause performance reduction
Squid	`cache_log`	Enable cache_log with higher debug_option would cause extra I/O
Squid	`store_objects_per_bucket`	Higher objects per bucket would enlarge the search time

Table 5: Unknown perf. effect of 11 parameters Violet identifies.

and running tests to confirm, which takes significant time. We only carefully inspect a subset of the results.

The four systems are very mature and maintain high-quality documentations, so it is not easy to find many errors in them. Indeed, a significant portion of the poor states we examined turns out to be already documented. Still we have identified 11 parameters that have potential bad performance effect and the documentation is incomplete or incorrect.

Table 5 lists the cases. For example, our analysis of `vacuum_cost_delay` shows that a higher value can incur large cost for write-intensive workloads, but the default value is 20 ms. Interestingly, we find PostgreSQL 12 (our experiments use v11) changes the default to 2 ms. For `log_statement`, Violet discovers multiple poor states that are not mentioned in the official document. Our analysis revels that setting it to `mod` causes performance issues for write query when `synchronous_commit` is `off`. Violet finds some unexpected parameter combination that leads to bad performance, e.g., `parallel_leader_participation` and `random_page_cost`.

We reported our findings to the developers. Eight reports are confirmed. Five lead to documentation or Wiki fixes. For some confirmed cases, developers do not fix them because they assume users should know the performance implications or such performance description should not be put in the reference manual (e.g., *"There are a lot of interactions between settings, and mentioning all of them would be impossible"*).

7.5 User Study on Violet Checker

To understand whether Violet checker helps users catch specious configuration, we conduct a controlled user study with 20 programmers (no authors are included). Fourteen are undergraduate students who have taken the database class. Six are graduate students. They all have decent experience with databases and Unix tools. We further give a tutorial of MySQL and PostgreSQL, the descriptions of the common configuration, and available benchmark tools they can use.

We use 6 target parameters from MySQL and PostgreSQL. For each parameter, we prepare two versions of configuration files. In one version (bad), the parameter is set with the poor value and the related parameters are also set appropriately that would cause bad performance impact under a workload. In another version (good), we set the target parameter to a good value, or we change the related parameter values, or we tell users the production workloads are limited to certain types (e.g., read-intensive). So in total, we have 12 cases.

Each participant is given 6 configuration files. They need to make a judgment regarding whether the configuration file would cause potential performance issue. Since a configuration file contains many parameters, we explicitly tell users the set of parameters they can focus on, which disadvantages Violet because users in practice do not have this luxury.

The participants are randomly assigned into two groups: *group A* (w/ Violet checker help) and *group B* (w/o checker help). Users in group B can run any tools to help them make the decision. We also tell group A users that they do not have to trust the checker output and are free to run other tools.

Figure 12 shows the accuracy of user study result for each group. Overall, programmers w/o Violet checker's help have 30% misjudgment rate while programmers with Violet checker's help only have 5% misjudging rate. Figure 13 shows the time for making a judgment. On average, participants took 20.7% less time (9.6 min. versus 12.1 min.) to make a judgment when they were provided with Violet checker. The reason that time saving is not very large is partly because we explicitly tell users the set of parameters, which creates a biased advantage to group B users; and some of our group A users are extra cautious and spend time running other tools.

7.6 Coverage of Analyzed Configs

We conduct a coverage test of Violet by applying Violet on the four software and try to derive performance models for as many parameters as possible. We manually filter the parame-

MySQL	PostgreSQL	Apache	Squid	Total
169 (51.2%)	210 (71.4%)	51 (29.6%)	176 (53.3%)	606 (53.9%)

Table 6: Number of configs Violet derives performance models for. The number in parentheses is the percentage of total configs.

	parA		parB		parC			parD		
	=0	=1	=0	=1	=0	=1	=2	=0	=1	=2
Violet	12.0	23.0	9.81	10.19	9.05	10.92	10.74	4.68	4.77	5.27
S^{2}E	10.8	21.0	7.67	8.94	6.24	7.77	7.92	3.57	3.91	4.59
Native	0.7	1.2	0.55	0.77	0.45	0.63	0.67	0.07	0.07	0.08

Table 7: Absolute latency (ms) for four parameters' different settings w/ Violet, vanilla S^{2}E and native execution. parA: `autcommit`, parB: `synchronous_commit`, parC: `archive_mode`, parD: `HostNameLookup`.

ters that are not related to performance based on the parameter description (*e.g.*, `listen_addresses`). Table 6 shows the result. Violet successfully derives models for a total of 606 parameters. The average ratio of analyzed parameters over the total number of parameters for software is 53.9%. The average number of states explored in these generated models is 23. Apache and Squid have a relatively small number of parameters analyzed. This is because the configuration program variables in the two systems are set via complex function pointers and spread in different modules, which make it challenging to write hooks to enumerate all of them (Section 4.1). For parameters that Violet did not generate impact models, one reason is that they are used in code for special environment. Another reason is that the data type of some parameter is too complex (*e.g.*, timezone) to make symbolic.

7.7 Accuracy of Violet Profiling

Since symbolic execution can introduce significant overhead, it seems that the latency traced by the symbolic engine will not be accurate. However, we observe that while the absolute latency under symbolic execution is indeed much larger than native execution, the comparative results between different paths are usually similar. We add a micro-benchmark experiment to test the latency measurement from Violet, vanilla S^{2}E and native mode. Table 7 shows the result from four representative parameters. Take parA as an example. The latency results from Violet and S^{2}E are much later than native result. But the ration of setting 1 to setting 0 is similar: 1.92× for Violet, 1.94× for S^{2}E, and 1.71× for native execution.

7.8 False Positives

The Violet differential performance analysis in general can absorb the performance noises in symbolic execution. But we observe some false positives in the Violet performance analysis output. For example, S^{2}E somehow has a delay in emitting the return signal of some system call functions like `gettimeofday`, which causes Violet to record inaccurate latency. These false positives are relatively easy to suppress by discounting the cost of the noisy instructions.

We manually inspect the performance models of 10 random parameters that Violet analyzes in the coverage experiment.

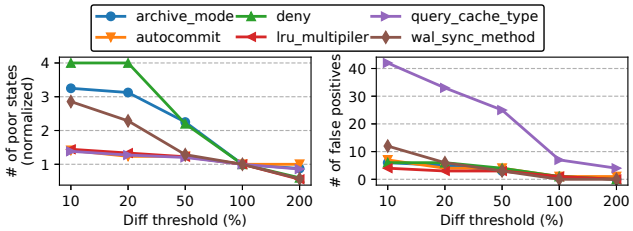


Figure 15: Sensitivity of the performance diff threshold (default 100%). For readability, the number of poor states is normalized by values under the default threshold.

We check the accuracy of the reported bad states by verifying them with `sysbench`. The false positive rate is 6.4%.

7.9 Performance

We measure the Violet analysis time for the 471 parameters in the coverage experiment (Section 7.6). Figure 14 shows the result in boxplots. The median analysis times are 206 s (MySQL), 117 s (PostgreSQL), 1171 s (Apache), and 554 s (Squid). On average, the log analyzer time is 68s. As explained in Section 7.3, even though for some parameters the analysis time is relatively long, the benefit is that Violet derives a thorough performance model for different settings of the target parameter and the combined effect with other parameters and input. The outcome can be re-used many times by the Violet checker. With the performance models, the checker time is fast. On average the checking only takes 15.7 seconds.

7.10 Sensitivity Analysis

Violet uses a differential threshold (default 100%) to detect the suspicious state from the trace log (Section 4.6). We evaluate the sensitivity of this threshold by measuring how many poor state pairs Violet reports when analyzing a parameter under threshold t. For each poor state pair Violet reports, we run benchmarks on the native machine to check whether it is false positive (performance difference is $\geq t$%).

Figure 15 shows the result for six representative parameters. We can see that if the threshold is set to a relatively lower value, the number of detected specious configuration can dramatically increase, but at cost of higher false positives.

8 Limitations

Violet has several limitations that we plan to address in future work. First, Violet explores the configuration under normal conditions. Some specious configuration may be only used in error handling. Exploring their effect requires specific faults. One solution is to combine symbolic execution with fault injection. Another potential solution is to use under-constrained symbolic execution [46]. Second, our handling of floating point type parameters is imperfect due to limited support in existing symbolic execution engines. We currently explores float parameters by choosing from a set of concrete floating-point values in the valid value range. Third, we use concrete (the host) hardware in the symbolic execution, which may not

capture specious configuration that is only visible in specific hardware. We rely on logical cost metrics to surface such issues. Lastly, Violet does not work on distributed systems.

9 Related Work

Misconfiguration detection and diagnosis. A wide body of work has been done to detect and troubleshoot misconfiguration [20–22, 27, 30, 30, 48, 50, 52, 54, 61, 63]. For example, ConfAid [21] uses dynamic taint tracking to locate configuration errors that lead to failures; Strider [52] and PeerPressure [50] take statistical approaches to identify misconfiguration; EnCore [63] enhances statistical learning with environment information to detect misconfiguration.

These solutions mainly target illegal configuration and have limited effects on specious configuration. X-ray [19] targets performance-related misconfiguration. Our work is inspired by X-ray and is complementary to it. X-ray is a diagnosis tool and uses deterministic record and replay of a specific program execution. Violet focuses on detecting specious configuration beforehand. Violet uses symbolic execution to explore the performance effect in multiple execution paths. Violet is more suitable for performance tuning/bug finding, whereas X-ray is better at diagnosing misconfiguration that has occurred.

LearnConf [41] is recently proposed to detect performance misconfiguration using static analysis. LearnConf summarizes common code patterns of performance configuration and uses simple formulas to approximate the performance effect, *e.g.*, linear relationship. It uses static analysis to identify these patterns and derive parameters to the formulas. The solution is simpler compared to Violet, but its completeness is limited because obtaining comprehensive code patterns is hard. Moreover, the performance effect is often quite complex, which cannot be accurately captured by simple formulas. Static analysis also suffers from well-known inaccuracies for large software. Violet explores a configuration's influence in the code without requiring or being limited by common patterns; it analyzes the performance effect by executing the code. Additionally, Violet explores the performance impact of input and a large set of related configurations together.

Performance tuning of configuration. There is a wealth of literature on automatic performance tuning, e.g., [33,44,51,55, 59, 62, 64]. They work basically by devising an approximate function between configuration values and the performance metrics measured through testing. While tunable parameters are common specious configuration, performance tuning and detecting specious configuration are two directions. The former searches for settings that yield the best performance, while the latter identifies settings that lead to extremely poor performance. Violet takes an analytical approach to derive configuration performance impact model from the code, instead of exhaustive testing. The result from our in-vivo, multi-path analysis is also less susceptible to noises and enables extrapolation to different contexts.

System resilience to misconfiguration. ConfErr [37] uses a human error model to inject misconfiguration. SPEX [57] uses static analysis to extract configuration constraints and generates misconfiguration by violating these constraints. The injected misconfigurations are illegal values that can trigger explicit errors like crash. Specious configuration typically does not cause explicit errors.

Configuration languages. Better configuration languages can help avoid misconfiguration. Several works make such efforts [23, 25, 28, 29, 35, 42, 47]. PRESTO [29] proposes a template language to generate device-native configuration. ConfValley [35], proposes a declarative validation language for generic software configuration. These new designs do not prevent specious configuration from being introduced.

Symbolic execution in performance analysis. Symbolic execution [24, 38] is typically used for finding functional bugs. S^2E [26] is the first to explore performance analysis in symbolic execution as one use case to demonstrate the generality of its platform. The Violet tracer leverages the advances made by S^2E, particularly its low-level signals, to build our custom profiling methods (Section 4.5). Our tracer also addresses several unique challenges to reduce the performance analysis overhead (Section 5.3). Bolt [36] extracts performance contracts of Network Function code with symbolic execution. Violet targets general-purpose software and analyzes performance effect of system configuration.

10 Conclusion

Specious configuration is a common and challenging problem for production systems. We propose an analytical approach to tackle this problem and present a toolchain called Violet. Violet uses symbolic execution and program analysis to systematically reason about the performance effect of configuration from code. The derived configuration performance impact model is used for subsequent detections of specious configuration. We successfully apply Violet on four large system software and detect 15 out of 17 real-world specious configuration cases. Violet exposes 11 unknown specious configuration, 8 of which are confirmed by developers.

Acknowledgments

We would like to thank our shepherd, Jason Flinn, and the anonymous OSDI reviewers for their thoughtful comments. We appreciate the discussion and suggestions from Xi Wang. We thank Varun Radhakrishnan and Justin Shafer for their contributions to the Violet tool and study cases. We thank our user-study participants and the open-source developers who responded to our requests. We also thank the S^2E authors, especially Vitaly Chipounov for maintaining the S^2E platform and answering our questions. We thank Chunqiang Tang for the prior collaboration that provided early motivation for this work. This work is supported by the NSF CRII grant CNS-1755737 and partly by NSF grant CNS-1910133.

References

[1] Amazon AWS S3 outage for several hours on February 28th, 2017. https://aws.amazon.com/message/41926.

[2] Amazon EC2 and RDS service disruption on April 21st, 2011. http://aws.amazon.com/message/65648.

[3] AWS service outage on October 22nd, 2012. https://aws.amazon.com/message/680342.

[4] Database administrators. https://dba.stackexchange.com.

[5] Facebook global outage for 2.5 hours on September 23rd, 2010. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919.

[6] Google API infrastructure outage on April 30th, 2013. http://googledevelopers.blogspot.com/2013/05/google-api-infrastructure-outage_3.html.

[7] Google compute engine incident #16007. https://status.cloud.google.com/incident/compute/16007?post-mortem.

[8] Google service outage on January 24th, 2014. http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html.

[9] Microsoft Azure storage disruption in US south on December 28th, 2012. http://blogs.msdn.com/b/windowsazure/archive/2013/01/16/details-of-the-december-28th-2012-windows-azure-storage-disruption-in-us-south.aspx.

[10] Microsoft Azure storage disruption on February 22nd, 2013. http://blogs.msdn.com/b/windowsazure/archive/2013/03/01/details-of-the-february-22nd-2013-windows-azure-storage-disruption.aspx.

[11] Oss-fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz.

[12] Percona blogs. https://www.percona.com/blog.

[13] RDS MySQL insights: Top query "commit". https://serverfault.com/questions/1029595/rds-mysql-insights-top-query-commit.

[14] Serverfault. https://serverfault.com.

[15] Slow InnoDB insert/update. https://www.serveradminblog.com/2014/01/slow-innodb-insertupdate/.

[16] Sysbench. https://github.com/akopytov/sysbench.

[17] Cisco loses customer data in Meraki cloud muckup due to misconfiguration. https://www.theregister.co.uk/2017/08/06/cisco_meraki_data_loss, Aug 6th, 2017.

[18] Amazon. AWS service outage on December 24th, 2012. http://aws.amazon.com/message/680587.

[19] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, 2012.

[20] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC'08, pages 281–286, 2008.

[21] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–11, 2010.

[22] L. Bauer, S. Garriss, and M. K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, pages 185–194, 2008.

[23] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 328–341, Florianopolis, Brazil, 2016.

[24] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, San Diego, California, 2008.

[25] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative configuration management for complex and dynamic networks. In *Proceedings of the 6th International Conference*, Co-NEXT '10, pages 6:1–6:12, 2010.

[26] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, Newport Beach, California, USA, 2011.

[27] T. Das, R. Bhagwan, and P. Naldurg. Baaz: A system for detecting access control misconfigurations. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 11–11, 2010.

[28] J. DeTreville. Making system configuration more declarative. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems*, HOTOS'05, pages 11–11, 2005.

[29] W. Enck, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, S. Rao, and W. Aiello. Configuration management at massive scale: System design and experience. In *Proceedings of the 2007 USENIX Annual Technical Conference*, ATC'07, pages 6:1–6:14, 2007.

[30] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, NSDI'05, pages 43–56, 2005.

[31] Google. Twilio billing incident post-mortem: Breakdown, analysis and root cause. https://www.twilio.com/blog/2013/07/billing-incident-post-mortem-breakdown-analysis-and-root-cause.html.

[32] J. Gray. Why do computers stop and what can be done about it? In *Proc. Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[33] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *In CIDR*, pages 261–272, 2011.

[34] Y. Hu, G. Huang, and P. Huang. Automated reasoning and detection of specious configuration in large systems with symbolic execution (technical report). http://arxiv.org/abs/2010.06356, 2020.

[35] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou. ConfValley: A systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 19:1–19:16, Bordeaux, France, 2015.

[36] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea. Performance contracts for software network functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 517–530, Boston, MA, USA, 2019.

[37] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings of the 38th International Conference on Dependable Systems and Networks*, DSN'08, pages 157–166, 2008.

[38] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[39] N. Kushman and D. Katabi. Enabling configuration-independent automation by non-expert users. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–10, 2010.

[40] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO '04, pages 75–, Palo Alto, California, 2004.

[41] C. Li, S. Wang, H. Hoffmann, and S. Lu. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, Heraklion, Greece, 2020.

[42] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 289–300, 2005.

[43] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, Mar. 2003.

[44] T. Osogami and T. Itoko. Finding probably better system configurations quickly. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 264–275, Saint Malo, France, 2006.

[45] A. Rabkin and R. Katz. How Hadoop clusters break. *IEEE Softw.*, 30(4):88–94, July 2013.

[46] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 49–64, Washington, D.C., 2015.

[47] A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter. A declarative language approach to device configuration. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11. ACM, March 2011.

[48] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 237–250, 2007.

[49] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 328–343, Monterey, California, 2015.

[50] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 17–17, 2004.

[51] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 154–168, Williamsburg, VA, USA, 2018.

[52] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th USENIX Conference on System Administration*, LISA '03, pages 159–172, 2003.

[53] X. Wei, S. Shen, R. Chen, and H. Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, ATC 17, pages 335–347. USENIX Association, July 2017.

[54] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 6–6, 2004.

[55] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 287–296, New York, NY, USA, 2004.

[56] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proceedings of the The 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, November 2016.

[57] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 244–259, 2013.

[58] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, San Jose, California, USA, 2011.

[59] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 196–205, San Diego, CA, USA, 2003.

[60] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, 2011.

[61] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ATC'11, pages 28–28, 2011.

[62] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, and et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, Amsterdam, Netherlands, 2019.

[63] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 687–700, 2014.

[64] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. BestConfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 338–350, Santa Clara, California, 2017.

Testing Configuration Changes in Context to Prevent Production Failures

Xudong Sun*, Runxiang Cheng*, Jianyan Chen, Elaine Ang, Owolabi Legunsen†, Tianyin Xu
University of Illinois at Urbana-Champaign †Cornell University

Abstract

Large-scale cloud services deploy hundreds of configuration changes to production systems daily. At such velocity, configuration changes have inevitably become prevalent causes of production failures. Existing misconfiguration detection and configuration validation techniques only check configuration values. These techniques cannot detect common types of failure-inducing configuration changes, such as those that cause code to fail or those that violate hidden constraints.

We present ctests, a new type of tests for detecting failure-inducing configuration changes to prevent production failures. The idea behind ctests is simple—connecting production system configurations to software tests so that configuration changes can be tested in the context of code affected by the changes. So, ctests can detect configuration changes that expose dormant software bugs and diverse misconfigurations.

We show how to generate ctests by transforming the many existing tests in mature systems. The key challenge that we address is the automated identification of test logic and oracles that can be reused in ctests. We generated thousands of ctests from the existing tests in five cloud systems.

Our results show that ctests are effective in detecting failure-inducing configuration changes before deployment. We evaluate ctests on real-world failure-inducing configuration changes, injected misconfigurations, and deployed configuration files from public Docker images. Ctests effectively detect real-world failure-inducing configuration changes and misconfigurations in the deployed files.

1 Introduction

1.1 Motivation

Large-scale cloud and Internet services evolve rapidly and deploy hundreds to thousands of configuration changes to production systems daily [35, 38, 53, 55]. For example, at Facebook, thousands of configuration changes are committed daily, outpacing the frequency of code changes [55]. Other cloud services such as Google and Azure also frequently deploy configuration changes [9, 10, 38].

The high velocity of configuration changes has led to prevalent configuration-induced failures. For example, faulty configurations are the second largest cause of service disruptions in a main Google production service [5]. At Facebook, 16% of service-level incidents, including major outages [54], are induced by configuration changes [55]. Similar levels of severity and prevalence of configuration-induced failures occur in other cloud systems [19, 34, 40, 42, 74].

Based on our experience from analyzing hundreds of configuration-induced incidents, failure-inducing configuration changes are rarely caused by trivial mistakes (e.g., typos). This rarity is attributed to the DevOps practices that enforce change review and validation [6, 27, 55]. As a result, the root causes of configuration-induced failures are often non-trivial; they commonly reside in the program and not in the changed configurations. Failures typically occur when *valid* configuration changes expose dormant software bugs [55] and when configuration changes violate undocumented, hidden configuration constraints. The root causes of the former are in the program, while the latter are often due to configuration design or implementation flaws [69]. Review and validation of configuration changes alone can hardly detect failures resulting from these root causes.

Researchers have proposed several configuration validation and misconfiguration detection techniques [70]. These include new languages and frameworks for implementing validators [6, 27, 55], detection techniques that use machine learning and document analysis to infer correctness rules on configuration values [38, 43, 44, 49, 50, 59, 61, 75, 77], and type or constraint checkers [48, 67]. These techniques are successful, but they are limited:

- Existing techniques only check configuration values and cannot detect configuration changes that cause code to fail.
- Very few existing techniques can detect "legal misconfigurations" [71], which have syntactically and semantically valid values but result in unexpected behavior.
- It is costly and hard for human-written or machine-learned rules to check the often subtle, version-specific [78], and inconsistent [69] configuration requirements.

1.2 Contributions

We present ctests, a new type of tests for detecting failure-inducing configuration changes to prevent production failures.

*Co-primary authors

Ctests take a simple and effective approach—connecting software tests with production system configurations. In this way, ctests can test configuration changes in the context of code that is affected by the changes. A ctest is parameterized by a set of system configuration parameters. Running a ctest instantiates each of its input parameters with a configuration value from production or a value to be deployed to production. Like regular software tests, ctests exercise system code and assert that program behavior satisfies certain properties (correctness, performance, security, etc). Ctests can be unit, integration, or system tests.

Existing software testing techniques do not connect tests to *actual* production system configurations. Rather, existing testing techniques sample *possible* configurations through systematic or random exploration of the enormous space of configuration value combinations [37]. Systematic exploration can be prohibitively expensive due to combinatorial explosion [39], while random exploration can have a low probability of covering all offending values that can cause production failures. Ctests have neither the cost of systematic exploration nor the low coverage of random exploration. By connecting tests to production system configurations, ctests can effectively detect failure-inducing configurations.

Ctests can test entire system configurations or incremental configuration changes in the form of configuration file "diffs." Our ctest infrastructure (see §3) supports *selectively* running only the ctests that are relevant to a configuration change, instead of re-running all ctests. Selectively running ctests saves testing time—most real-world configuration changes modify only a few configuration values [55].

We show how to generate ctests by transforming the existing and abundant tests in mature software projects in an automated fashion that reuses well-engineered test logic and oracles. The main challenge that we address is the automated identification of test logic and oracles that can be transformed into ctests. Existing test logic may assume specific configuration values. Such assumptions can be implicit (assuming default values) or explicit (hardcoding certain values). Thus, naïve parameterization will not always generate valid ctests.

Our transformation identifies and respects the intent of existing tests that assume specific configuration values. First, configuration parameters whose values are explicitly reassigned in the test code are excluded from the input parameter set of a ctest. Then, the values of the parameters used in candidate ctests are varied to observe the corresponding test output. We exclude parameters whose values are hardcoded in a test because such tests will fail on different but valid values. Our tests-to-ctests transformation is mechanized in a toolchain and we successfully generated 7,974 ctests by transforming the existing test suites in five cloud systems.

Ctests address the following limitations of existing configuration validation and misconfiguration detection techniques:

- Ctests can detect failure-inducing configuration changes where the root cause of the failure is in the code.
- Ctests can detect legal misconfigurations by capturing the resulting unexpected system behavior.
- Ctests can be generated from existing tests, without incurring the high cost of learning or codifying rules.

Our results show that ctests can effectively detect failure-inducing configuration changes before deploying them to production. We evaluated ctests using 64 real-world configuration-induced failures, 1,055 diverse misconfigurations generated by error injection rules, and 92 deployed configuration files from publicly-available Docker images.

Ctests detected the failure-inducing configurations in 96.9% of the real-world failures. The ctests that detected these real-world failures were transformed from the tests in the older version of the systems on which the failures were reported. That is, ctests could have detected these failures earlier. Ctests also detected 10 misconfigurations in 7 deployed files. Additionally, our ctest generation process exposed 14 previously unknown bugs, including a bug that users encountered after we reported it [24]. Developers confirmed 12 of these 14 bugs and fixed 10 of them.

In summary, this paper makes the following contributions:

- Ctests enable a simple and effective approach for detecting failure-inducing configuration changes.
- We present how to generate ctests by transforming the many existing tests in mature systems.
- We show that ctests can effectively detect real-world configuration-induced failures early, during testing.

2 Background and Examples

We describe how ctests address the limitations of existing techniques for validating configuration values [6, 27, 48, 55, 67] and techniques for detecting specific types of misconfigurations [38, 43, 44, 49, 50, 59, 61, 62, 75, 77].

Checking configurations based on program behavior. A key capability of ctests is to check how actual configuration values impact program behavior. This capability is essential for detecting configuration changes that result in code failures or expose dormant bugs. In our experience, checking program behavior can be more effective in capturing failure-inducing configuration changes than checking configuration values against rules (which are usually incomplete).

Figure 1 uses a real-world issue from HBase [21] to illustrate the capability of ctests. There, a ctest detects a misconfiguration that degrades performance ("*too many handlers can be counter-productive* [56]"). The ctest is generated from an existing test in the reported HBase version. It asserts on the computed schedule with the expected behavior that handler counts are not affected by configuration changes. The offending value is "legal" [71] but the reported version had no validation code to check the expected behavior.

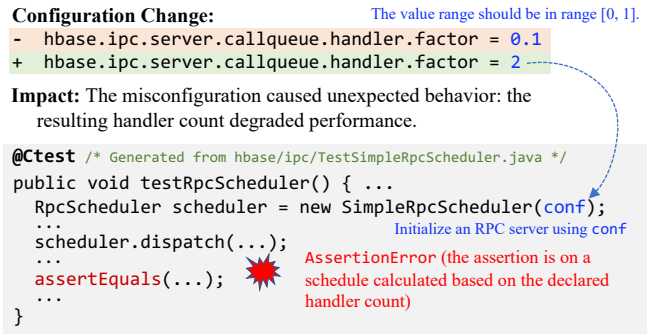


Figure 1: A ctest that detects a real-world misconfiguration in HBase [21] by checking expected system behavior. The ctest is generated from a test available in the reported HBase version.

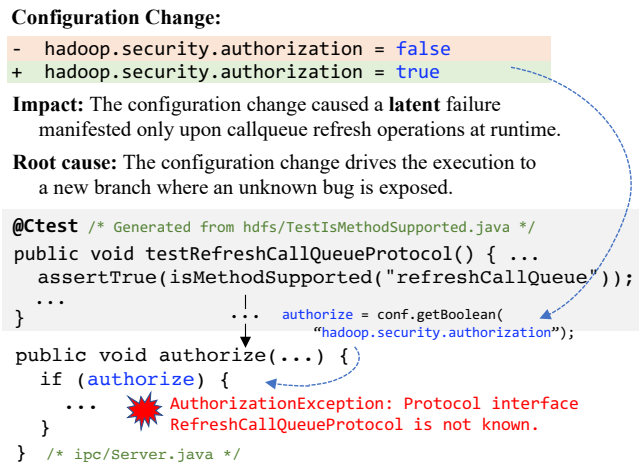


Figure 2: A ctest that detects a dormant bug exposed by a configuration change in Hadoop [20]. The ctest is generated from a test available in the reported Hadoop version.

Detecting dormant bugs exposed by valid configuration changes. Ctests can detect not only misconfigurations but also software bugs exposed by valid configuration changes. Such bugs are common root causes of configuration-related incidents (§1.1). Existing configuration validation and misconfiguration detection techniques only check for erroneous configuration values; they are fundamentally limited to detecting failures with root causes outside the changed configurations. Such software bugs inevitably occur, despite extensive testing and static analysis. Some bugs can only be exposed under specific configurations. Figure 2 shows a real-world failure from Hadoop [20]. A failure-inducing configuration change caused Hadoop to traverse new execution paths and exposed a dormant software bug.

Detecting diverse misconfigurations. Many existing techniques focus on detecting specific kinds of misconfigurations. Ctests are generic. They can detect configuration changes that lead to any kind of unexpected program behavior. So ctests can detect misconfigurations that are hard for state-of-the-art techniques to detect. Such misconfigurations involve (1) custom regular expressions, user commands, and URIs (statistical analyses and machine learning can detect outliers but cannot deal with custom values [67]), (2) invalid content referred to by path-related configurations (most existing techniques only check metadata), (3) violations of undocumented constraints that cannot be found by text-based document analysis [44, 59, 65], and (4) dependencies among multiple configuration parameters [12]. Figures 8 and 9 show examples of misconfigurations detected by ctests that are hard to detect using existing techniques.

Incremental pre-deployment testing for every configuration change. Ctests can help *prevent* failure-inducing configuration changes from being deployed to production systems. The goal of ctests is to test every configuration change early, during testing. Ctests can be run selectively on configuration file "diffs" to save testing time (§3.2). Ctests do not suffer from limitations of post-deployment configuration checking (e.g., disallowing operations with side effects to avoid corrupting production system states as in PCheck [67]).

3 Ctest Overview

The idea behind ctests is to connect production system configurations to software tests, enabling the checking of configuration changes against program properties in the context of code affected by the configuration changes. Ctests detect *both* misconfigurations caused by assigning invalid values to configuration parameters and bugs in the code that are exposed by changing configuration parameters to new valid values.

3.1 Ctest Definition

A ctest, $\hat{t}(\hat{P})$, is a test $\hat{t}$ that is parameterized by a set of system configuration parameters $\hat{P}$. Running a ctest instantiates each parameter $p \in \hat{P}$ with a concrete value. In particular, each $p \in \hat{P}$ in a ctest can be instantiated with a value from the production system configuration or a configuration change (in the form of a configuration file "diff") to be deployed. Note that $\hat{P}$ is typically only a very small subset of all system configuration parameters, denoted as $\mathbb{P}$. That is, $|\hat{P}| \ll |\mathbb{P}|$.

Ctests can be unit, integration, or system tests. Like regular software tests, a ctest can assert on different kinds of program properties: correctness, performance, security, etc. Ctests can be written from scratch by developers, or they can be generated from existing software tests (see §4). Our generation procedure reuses test logic and assertions in existing tests during transformation to ctests.

3.2 Ctest Usage

Ctests can check an entire system configuration, a configuration change, or a configuration file. So, ctests can be used both as a traditional configuration file checker and as an enabler of configuration checking during continuous integration and

deployment [52]. Ctests are complementary to configuration validation, similar to how software testing complements static analysis for bug detection.

Ctests for checking entire system configurations. We define a system configuration as the values of all configuration parameters in the system denoted as $C = \bigcup_{i=1..|\mathbb{P}|}\{(p_i \mapsto v_i)\}$ (it assigns a value v_i to every parameter p_i in $\mathbb{P}$). Running a ctest, $\hat{t}(\hat{P})$, instantiates each parameter $p_i \in \hat{P}$ with its value in the system configuration v_i such that $(p_i \mapsto v_i) \in C$. To test the system configuration, C, all available ctests are run. C passes if all ctests pass and fails if any ctest fails.

Ctests for checking configuration changes. In modern continuous integration and deployment, a configuration change has the form of a configuration file *"diff"*. A diff typically only changes the values of a small set of configuration parameters, P_D [55]. It updates the system configuration from C to C' by changing each $p_d \in P_D$'s value from v_d to v'_d. Formally, we define a configuration diff $D = \{(p_d \mapsto v'_d) \mid p_d \in P_D \text{ and } (p_d \mapsto v_d) \in C \text{ and } v_d \neq v'_d\}$.

For a given diff D, a ctest $\hat{t}(\hat{P})$ can be used to test D if at least one configuration parameter in D is in its input parameter set $\hat{P}$ (i.e., if $P_D \cap \hat{P} \neq \emptyset$). We use this *test selection criterion* to re-run only the subset of ctests whose outcome could be altered by D, instead of re-running all ctests after every configuration change.

A selected ctest $\hat{t}(\hat{P})$ can be run before deploying D to production by assigning values in D to the ctest's parameters that are in D and assigning values in C to the ctest's parameters that are not in D. Precisely, assign v'_d to each $p_d \in \hat{P} \cap P_D$, where $(p_d \mapsto v'_d) \in D$; then, assign v to each $p \in \hat{P} - P_D$, where $(p \mapsto v) \in C$. Ctests with $\hat{P} \cap P_D = \emptyset$ do not need to be run when testing D. A configuration diff, D, passes if all selected ctests pass and fails if any selected ctest fails.

Ctests for checking configuration files. A configuration file typically only assigns values to a subset of $\mathbb{P}$. Parameters whose values are not assigned in the configuration file receive their default values. So, ctests treat a configuration file as a diff which updates the default system configuration with the configuration values that are set in the file.

Locating offending configuration values. If a ctest is newly failing on a configuration diff, D, then the offending parameters must be in $\hat{P} \cap P_D$, unless the tests are flaky [8]. Parameters in D are typically very few, e.g., 49.5% of configuration changes have two-line revisions [55]. We discuss our experience on inspecting ctest failures in §7.

3.3 Creating a Ctest Infrastructure

Ctest infrastructure can be built on top of existing software testing frameworks. Specifically, a ctest can be run in the same way as a regular software test by instantiating the test's input parameters with system configuration values. We built our current ctest infrastructure on top of the Maven build system [36]—all the systems that we study use Maven to compile and run their test suites (§5.1). It should be straightforward to extend our infrastructure to support other build systems such as Gradle [16], Bazel [7], and Buck [11].

Ctests should be run in a hermetic test environment (a common software testing practice [41]). Ctests are best run in the same environmental setup as in production because ctests can capture environment-specific, configuration-induced failures (e.g., Figure 8). Our current infrastructure supports running ctests in Linux containers.

Ctest selection. Ctest selection is critical for utilizing ctests during continuous integration and deployment of configuration diffs. Regression test selection, which reruns tests that are affected by code changes [17], does not work for configuration changes. We build our ctest selection mechanism using the test selection criterion described in §3.2; it only runs ctests that are parameterized by parameters in D.

Configuration versioning. We store the latest version of the system configuration C to be updated after a configuration diff passes ctest and is deployed (§3.2). So, our infrastructure can instantiate ctests with updated parameter values in C.

4 Ctest Generation

Ctests can be generated by transforming existing tests in mature software projects with reasonable manual effort. The generated ctests inherit test logic and assertions from the original tests. The inherited assertions hold for *all* correct configuration values.

Ctest generation proceeds in two steps. First, the existing tests are *parameterized* by system configuration parameters, so that they can be run against different system configurations (§3.2). We describe in §4.1 how to parameterize an existing test t to obtain $\hat{t}(\mathbb{P})$ (or $\hat{t}$ in short), where $\mathbb{P}$ represents all the configuration parameters of the target system. Second, the parameterized tests are transformed into ctests.

A parameterized test $\hat{t}$ may not be directly usable as a ctest if the original test t contains test logic or oracles that assume specific configuration values. The resulting parameterized test $\hat{t}$ may fail incorrectly on valid configuration values if the subsequently resulting ctest is run against new values that are not the assumed values. So, if $\hat{t}$ assumes specific values of a configuration parameter $p \in \mathbb{P}$, $\hat{t}$ cannot be a ctest for p. But $\hat{t}$ can still be a ctest for another independent parameter, say $q \in \mathbb{P}$, if $\hat{t}$ does not assume a value for q. In short, if $\hat{t}$ assumes a value for p but not for q, $\hat{t}$ can result in a ctest for q but not for p. We address the challenge of identifying, among all configuration parameters exercised by $\hat{t}$, those that can be included in the input parameter set $\hat{P}$ of the resulting ctest $\hat{t}(\hat{P})$. In this example, $q \in \hat{P}$ and $p \notin \hat{P}$. We describe in §4.2 how to identify $\hat{P}$ from $\mathbb{P}$ when generating a ctest $\hat{t}(\hat{P})$ from $\hat{t}$.

One can optionally rewrite generated ctests to allow generated ctests check more configuration parameters or to generate

```
1     static {
2       ...
3       addDefaultResource("core-default.xml");
4       addDefaultResource("core-site.xml");
5   +   addDefaultResource("core-ctest.xml");
6     }
7     /* conf/Configuration.java */
```

Figure 3: Parameterization by intercepting the configuration APIs of Hadoop. After the interception, test code reads configuration values from `core-ctest.xml` which is managed by our ctest infrastructure. In this way, the test code can be instantiated with values in `core-ctest.xml`.

new ctests. §4.3 presents two simple rewriting rules for dealing with hardcoded parameter values and assertions.

In summary, given tests $T = \{t_i \mid i = 1, 2, ..., Nf\}$, we generate a set of ctests $\hat{T} = \{\hat{t}_i(\hat{P}_i)\}$, where $|\hat{T}| \leq |T|$. For each ctest $\hat{t}_i(\hat{P}_i)$, $\hat{t}_i$ is the parameterized test and $\hat{P}_i$ is the set of configuration parameters that can be tested by the ctest. Each ctest is generated from an existing test and checks one or more parameters. To test to-be-deployed configurations, a ctest instantiates all its input parameters.

Developer effort. To generate ctests from existing tests, developers need to instrument the configuration APIs of the system. We discuss instrumentation in §4.1 and §4.2.1. After instrumentation, ctest generation is mechanized.

4.1 Parameterization

The first step in generating ctests is to parameterize an existing test t into $\hat{t}$. so that t can be run by instantiating the parameters with actual system configuration values. Parameterization requires changing test code to read configuration values at runtime, as provided by ctest infrastructure (§3.3), instead of from default configuration files or other test files. To generate large numbers of ctests, parameterization is automated.

We find that systematic parameterization can be done by intercepting the configuration APIs that existing tests use for reading configuration values. Figure 3 exemplifies our interception of Hadoop's configuration API. The idea is to overwrite configuration values as the final step of configuration loading. Thus, when the test code reads configuration values from configuration APIs, the values come from the configurations maintained by the ctest infrastructure (§3.3). Our parameterization approach minimizes the changes needed and avoids changing individual tests. Our approach is applicable to many (if not all) modern cloud systems, but its implementation is project-specific. We implemented parameterization for five cloud systems (§5.1) and validated its applicability to other systems including Spark and OpenStack.

The parameterization step produces a parameterized test, $\hat{t}(\mathbb{P})$, for each test t, where $\mathbb{P}$ is the set of all system configuration parameters. Parameterization is oblivious of the set of configuration parameters exercised by each $\hat{t}$; these are automatically identified in §4.2.1.

4.2 Transformation

A parameterized test $\hat{t}(\mathbb{P})$ may not be a valid ctest—a ctest's parameter set $\hat{P}$ should include only configuration parameters that can be checked by the ctest—the test logic and oracles should not assume specific parameter values.

Transforming a parameterized test into a ctest consists of (1) identifying the set of configuration parameters that are exercised by each test t, denoted as P (§4.2.1), and (2) for each $p \in P$, determining whether the test logic and oracle of t assume any specific value of p; if so, $p \notin \hat{P}$ (§4.2.2). Figure 4 shows ctest generation process that transforms from t to $\hat{t}(\hat{P})$.

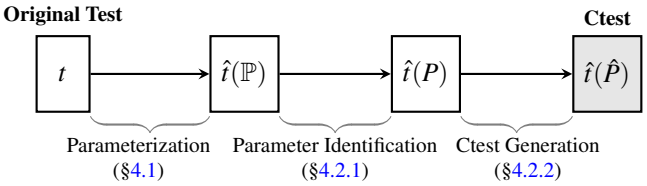


Figure 4: Steps in the ctest generation process.

4.2.1 Identifying Parameters Exercised in Tests

Static or dynamic analysis can be used to identify P for each test t. We implemented and experimented with both. Our static analysis taints the statements that can be reached by t and searches for configuration API usage (§4.1) among the tainted statements. It was straightforward to identify configuration API usages in test code. But, since test code commonly passes configuration values to system code initialization, it is hard to precisely collect the exact configuration API usage in system code that may be reachable from tests. So, static analysis often imprecisely produces a parameter set larger than P.

We chose dynamic analysis under the assumption that most test cases are relatively deterministic [15]. Our dynamic analysis requires developers to instrument configuration `GET` and `SET` APIs for reading and writing configuration values in the target system, respectively.[1] Our instrumentation inserts code to log the stack trace of each API invocation and the configuration parameter involved. Figure 5 is an example of our instrumentation for Hadoop. With instrumentation in place, our dynamic analysis runs all existing tests and post-processes the log for each test t to automatically identify (1) the set of configuration parameters P exercised by t, and (2) parameters written (via the `SET` API) in t (needed in §4.2.2).

Log processing is automated, as our instrumentation produces easily-parsed output. Our dynamic approach is simple,

[1] The `GET` and `SET` APIs are common configuration abstractions used in cloud systems written in Java and Python [33, 48, 67, 69]. `GET` APIs are of the form, "`<T> get(Class<T> class, String parameter)`"; they take a parameter name and return a value. `SET` APIs are of the form, "`void set(Class<T> class, String parameter, <T> value)`"; they reset the original value of the given parameter with the input `value`. Typically, `get` and `set` are declared in wrapper classes such as `java.util.Properties` for Java and `configparser` for Python projects.

```
1    public String get(String name) {
2  +   String ctestParam = name;
3      String[] names = handleDeprecation(
4          deprecationContext.get(), name);
5      String value = null;
6      for(String n : names) {
7  +     ctestParam = n;
8        value = substituteVars(
9            getProps().getProperty(n));
10     }
11 +   LOG.warn("[CTEST][GET-API] " + ctestParam);
12 +   CTestUtils.printStackTrace(ctestParam);
13     return value;
14   }
15   /* conf/Configuration.java */
```

Figure 5: Example Instrumentation for a `GET` API in HCommon. The `get` method is the lowest level API used by high-level `GET` APIs, e.g., `getInt` and `getBool`. `handleDeprecation` handles deprecated parameters. `SET` APIs are instrumented similarly.

general, and reliable, requiring modest instrumentation effort (§5.1). Instrumentation of configuration APIs is performed during ctest generation. Instrumentation is neither performed when running ctests nor added to the production system.

For completeness, we consider a test to *exercise* a parameter if the test uses the parameter's value as it executes. We do not exclude tests based on potential effectiveness for exposing misconfigurations or bugs. In general, such effectiveness is hard to define or model. Our decision is also justified because `GET` API invocations alone can expose misconfigurations (e.g., those due to type-casting errors [26]) or bugs (e.g., those caused by failing to trim white space [25]).

4.2.2 Generating Parameter Sets for Ctests

For each test $\hat{t}(P)$ that is parameterized after the steps in §4.1 and §4.2.1, our toolchain automatically generates the ctest $\hat{t}(\hat{P})$ by filtering out configuration parameters in $P - \hat{P}$:

Respecting intended configuration resets. If a test explicitly resets a configuration parameter to a specific value, then the test logic or its oracle depends on the new value. So, the test cannot be applied to other valid values of the configuration parameter. Our tool automatically identifies all configuration parameters whose values are reset in a test t. It does so by parsing the logs generated by instrumented `SET` APIs (§4.2.1) and excluding parameters that are reset from P. Note that we do not exclude configuration parameters from P that are reset in the system (not test) code. System code can reset configuration values in ways that should not impact ctests, e.g., during dynamic configuration tuning.

Detecting implicit assumptions on configuration values. In practice, not all parameter resets are performed using `SET` APIs. Some tests *implicitly* assume specific parameter values. Most tests with such implicit assumptions expect default parameter values and do not set them explicitly. If the default value is unchanged, then the tests pass. Although such assumptions constitute bad software engineering practice ("brittle assertions" in the literature [28]), we observe many such cases in the existing test code. Therefore, we automatically identify and exclude from $\hat{P}$ the parameters on which tests have implicit assumptions.

Our intuition is that, if a test assumes specific values, then it will fail on different but valid values. That is, if $p \in \hat{P}$, then $\hat{t}$ should pass on all valid values of p. So, a configuration parameter on which a test makes an implicit assumption can be identified by assigning a different valid value to the parameter and observing the outcome of the existing test.

Our implementation validates whether $\hat{t}$ makes implicit assumptions on each $p \in P$ by running $\hat{t}$ with p instantiated with a few valid values. If $\hat{t}$ fails on a valid value, then $\hat{t}$ makes an assumption on the value of p, i.e., $p \notin \hat{P}$. In our experience in generating thousands of ctests (§5), using up to three values for validation is sufficient to identify configuration parameters on which tests make implicit assumptions.

We use heuristics to automatically generate values for validation from the default value of each configuration parameter, based on the parameter types. For numeric types, we halve and double the original value. For Boolean values, we use the negation. For environment-related values (e.g., path, address, and port), we generate a similar but different value (e.g., a different port number). We use the regular expression described in [77] to infer parameter value types.

These heuristics are not sound; they do not guarantee the validity of generated values. However, the heuristics are simple and practical—only 1.6% of the generated values were invalid due to hidden constraints (§5.3). Our heuristics could not generate valid values for about 16% of parameters: enum options, class names, and commands. We manually selected valid values in these cases. Our future work includes integrating advanced inference tools [44, 47, 69] to infer valid values for these parameter types.

The validation yields $\hat{P}$ for each parameterized $\hat{t}$ transformed from t. If $\hat{P} \neq \emptyset$, $\hat{t}(\hat{P})$ is a ctest for all $p \in \hat{P}$.

4.3 Rewriting

In addition to the generated ctests, one can optionally manually rewrite an existing test to create a new ctest or rewrite a generated ctest to check more configuration parameters. We find two common patterns for rewriting, exemplified in Figure 6. First, many configuration resets in test code are used for setting up the test environment, e.g., a test file, address, port, etc. Those resets are not needed in ctests which are run with actual environment variables. Figure 6a shows this rewriting pattern. There, by simply removing the reset, the ctest can check `alluxio.master.rpc.port`'s values. Note that removing hardcoded resets may require changing how the test reads the configuration values, if the test code does not use standard APIs (discussed in §5.4). Second, some assertions in the test code assume the default configuration values (§4.2.2) which can be safely removed or rewritten to the actual values

```
1     @Ctest
2     void testStartStopPrimary() {
3   -   conf.set("alluxio.master.rpc.port",
4   -       TEST_PORT);
5       master = new AlluxioMasterProcess(conf);
6       master.start();
7       ...
8     }
9     /* master/AlluxioMasterProcessTest.java */
```

(a) Removing hardcoded resets. After removing the `conf.set()` call, the `alluxio.master.rpc.port` parameter's value comes from system configuration. The rewritten ctest can then test `alluxio.master.rpc.port`.

```
1     @Ctest
2     void testNameNodeXFrameOptionsEnabled() {
3       ...
4       header = conn.getHeaderField(
5           "X-FRAME-OPTIONS");
6       ...
7       assertTrue(header.endsWith(
8   -       HttpServer.XFrameOption.SAMEORIGIN));
9   +       conf.getTrimmed("dfs.xframe.value")));
10    }
11  /* namenode/TestNameNodeHttpServerXFrame.java */
```

(b) Rewriting hardcoded assertions. The rewritten ctest asserts on the actual value of the `dfs.xframe.value` parameter not its default value (`SAMEORIGIN`).

Figure 6: Two common patterns of test rewrites (§4.3).

being tested, as shown in Figure 6b. For both patterns, test code is rewritten to read values from the system configuration without changing the test logic.

5 Generating Thousands of Ctests

We share our experience in generating over 7900 ctests by transforming existing tests in five mature and widely-used open-source cloud systems: HCommon (Hadoop runtime and core utilities), HDFS, HBase, ZooKeeper, and Alluxio. We chose these projects for our evaluation (§6) because they are widely studied, their configuration APIs represent the state-of-the-art in modern cloud systems, and they expose many configuration parameters (Table 1). We discuss the feasibility of, and opportunities for, generating ctests in practice.

5.1 Evaluated Systems and their Test Suites

Table 1 shows the characteristics of the cloud systems that we studied: tests, configuration parameters, and how much instrumentation we performed.

Instrumentation effort. Our system-specific instrumentation is modest because each system uses a few classes to implement the configuration APIs. In the worst case, we changed only three classes each in ZooKeeper and Alluxio ("# Class" column in Table 1). It takes more lines of instrumentation for ZooKeeper than the others, because the `GET` and `SET` APIs

Software	Test Coverage		# Config.	Instrum.	
	Stmt Cov.	Meth Cov.	Params	LoC	# Class
HCommon (2.8.5)	73.1%	74.0%	269	24	1
HDFS (2.8.5)	80.3%	79.6%	296	24	2
HBase (2.2.2)	69.5%	80.1%	205	29	2
ZooKeeper (3.5.6)	75.8%	84.3%	32	130	3
Alluxio (2.1.0)	70.8%	72.6%	515	34	3

Table 1: Characteristics of studied systems (test suites, configuration parameters, and instrumentation efforts). The instrumentation includes both parameterization (§4.1) and logging (§4.2.1).

Software	Module	# Tests		# Config. Param.	
		Total	Using Config.	Total	Used in Tests
HCommon	hadoop-common	3268	1923 (58.8%)	269	232 (86.2%)
HDFS	hadoop-hdfs	3957	3293 (83.2%)	296	284 (95.9%)
HBase	hbase-server	2630	2035 (77.4%)	205	169 (82.4%)
ZooKeeper	zookeeper-server	881	180 (20.4%)	32	32 (100.0%)
Alluxio	core	1648	1117 (67.8%)	515	423 (82.1%)

Table 2: Characteristics of configuration parameters exercised in software tests of the studied systems.

are implemented per configuration parameter;[2] the other four systems implement generic APIs as exemplified in Figure 5.

Test suites. The studied systems all have a good number of tests, mostly at the unit- and integration-test levels. System-level tests are rare, reflecting a common testing practice in modern systems engineering [60]. Further, all five projects enforce rigorous code commit policies that require every code change to be covered by a test. Code coverage is high ("Test Coverage" in Table 1), with at least 70% statement coverage. Proprietary systems report even higher test coverage [29, 51].

We focus on the core modules of the studied systems ("Module" in Table 2), which are likely to be used in production. In the rest of this paper, we only use the tests in the studied modules, even though tests in the other modules can also be leveraged during ctest generation.

Table 2 shows the percentage of existing tests per module that exercise configuration values ("Using Config.") and the percentage of configuration parameters exercised by tests ("Used in Tests"). We collected these percentages from instrumented configuration API logs (see §4.2.1). Clearly, many tests exercise configuration values and are candidates that can be transformed into ctests. Furthermore, 82.1%–100.0% of configuration parameters across the studied systems are exercised by existing tests. So, most configuration parameters have a chance of being checked by a generated ctest (§5.2).

5.2 Ctest Generation Results

We apply the automated approach in §4.2 to generate ctests from the existing tests in the evaluated systems. We select all 32 configuration parameters in ZooKeeper. For the other

[2] We are helping ZooKeeper to improve their APIs (e.g., [81]); using ZooKeeper shows applicability of ctests across configuration APIs.

Software	Existing Tests			Generated Ctests	
	# Param.	Tests	→	Ctests	#Param. (Cov.)
HCommon	90	1870	→	1846 (98.7%)	90 (100.0%)
HDFS	90	3191	→	3148 (98.7%)	90 (100.0%)
HBase	90	1909	→	1687 (88.4%)	90 (100.0%)
ZooKeeper	32	180	→	176 (97.8%)	32 (100.0%)
Alluxio	90	1117	→	1117 (100.0%)	90 (100.0%)

Table 3: Ctest generation results. The results include only generated ctests (§4.2). The generated ctests have 100% coverage of the configuration parameters.

systems, we randomly select 90 configuration parameters that are exercised by the tests ("Used in Tests" in Table 2). Note that we sampled 90 parameters mainly to bound our manual inspection effort for analyzing effectiveness and false negatives (Tables 8 and 9). The generation process is mostly automated after API instrumentation.

Table 3 shows ctest generation results. Overall, 88.4%–100% of existing tests that exercise the selected configuration parameters were successfully transformed into ctests. Furthermore, the generated ctests cover 100% of the selected parameters, i.e., each parameter is checked by at least one ctest. The small percentage of tests that could not be transformed as ctests were hardcoded to specific values of *all* the parameters that they exercise—a ctest is generated as long as it can check at least one configuration parameter. Section 6 discusses the effectiveness of the generated ctests for detecting failure-inducing configurations in different settings.

5.3 Detecting Bugs and Hidden Constraints

Some *valid* configuration values caused ctests (§4.2.2) to unexpectedly throw runtime exceptions instead of the failed assertions that are typical manifestations of hardcoded tests. We analyze these exceptions and find, surprisingly, that most are caused by (1) previously unknown bugs in the code exposed by configuration changes, or (2) hidden constraints which made seemingly valid configuration values erroneous. We include these ctests which are effective in capturing bugs and misconfigurations in our evaluation.

Dormant bugs exposed by configuration changes. We find 14 previously unknown bugs in the latest versions of the five evaluated systems. 12 of those bugs are confirmed and 10 were fixed by the developers after we reported them; 9 bugs are considered "Major" or "Critical". Real users encountered a bug after we reported it [24]. 12 of the 14 bugs existed for more than five years in these projects that routinely run static analyses and perform testing. Figure 7 shows one of these bugs, in which changing the value of the parameter to a valid option `TopAuditLogger` will crash the NameNode of HDFS because a default constructor is required but not implemented.

Hidden configuration constraints. We also discovered 11 hidden constraints that cause the generated values to result in errors. We say these constraints are "hidden" because they

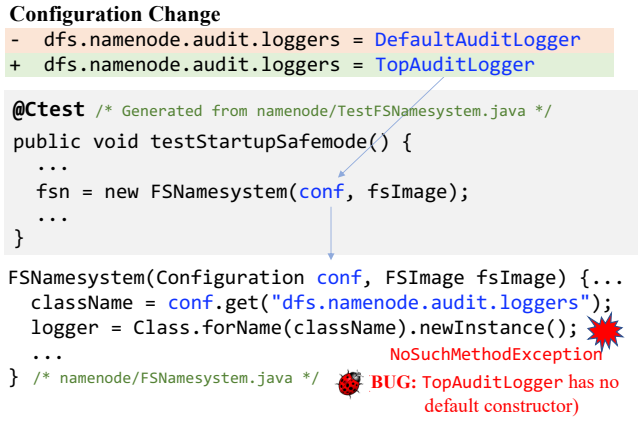


Figure 7: A new bug that was exposed by a *valid* configuration change and was captured by a ctest [23]. The bug crashes HDFS NameNode due to missing a default constructor. The bug has been fixed after we reported it.

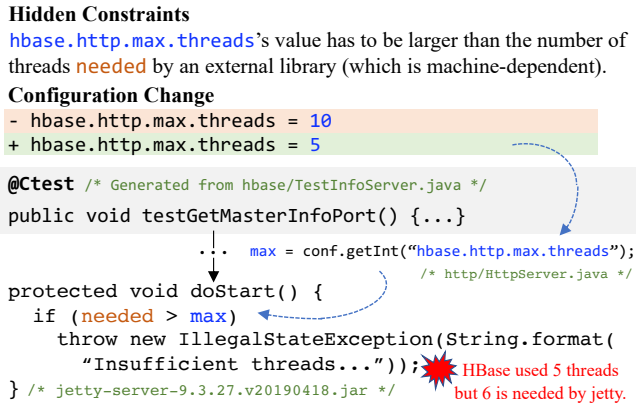


Figure 8: A hidden constraint exposed by a ctest. The configuration of HBase is constrained by an external library (Jetty).

were not documented and are not intuitive to discover. Figure 8 is an example of a hidden constraint—the configuration parameter of HBase is constrained by an external library, Jetty. Any configuration value that is smaller than the `needed` variable's value in Jetty will cause a runtime exception.

5.4 Rewriting Ctests

We further study the intended configuration resets in test code (§4.2.2) to understand the opportunities and challenges of rewriting tests. We focus on environment-related configuration parameters—as discussed in §4.3, tests often reset configuration values to set up test environments, which are not needed by ctests. For this study, we selected 44 configuration parameters with hardcoded environment settings, including all four from ZooKeeper and 10 from the other four systems. There are altogether 263 tests that reset at least one of the 44 parameters; 233 of these tests were transformed to generate ctests but those ctests cannot check the reset parameters. The 233 generated ctests cover all 44 parameters (Table 3).

We manually applied the two test rewriting rules described in Figure 6 to these 263 tests. Removing hardcoded resets alone (Figure 6a) can enhance 86 tests for ctests to cover 8 parameters. Further, by removing or rewriting hardcoded assertions (Figure 6b), we can enhance 16 more tests. In total, the two test rewriting rules can cover 102 (38.8%) tests for 18 out of 44 parameters. The remaining tests either cannot benefit from rewriting, or require significant changes beyond the two simple patterns in Figure 6.

The test rewriting effort was small in HCommon, HDFS, HBase, and Alluxio for which we rewrote 33 tests for 16 parameters using a total of 90 changed lines. Rewriting a test in these four systems takes only two or three lines of test code (Figure 6). The rewriting effort was much larger in ZooKeeper, mainly because ZooKeeper does not utilize similar configuration APIs (§5.1) as other systems—the test code does not use SET APIs to reset the parameter value as in Figure 6a. So, we wrote a new API to load actual configuration values into the tests; our implementation has 14 lines of code. With our new API, we were able to rewrite 69 tests for two parameters, which takes a total of 103 changed lines.

6 Evaluation of Ctest Effectiveness

We used three experimental settings to extensively evaluate ctests' effectiveness for testing configurations in context:

1. real-world configuration-induced failures documented in issue tracking databases;
2. diverse injected misconfigurations for configuration parameters that have different value types and semantics;
3. non-default configuration files collected from Docker images hosted at DockerHub [14].

6.1 Evaluating Ctests on Real-world Failures

We evaluate the effectiveness of ctests for detecting failure-inducing configurations that caused real-world failures. Our goal is to see how many of these failures ctests could have been detected earlier.

Configuration-induced failures used. We reproduced 64 real-world configuration-induced failures from the issue-tracking database of the five systems (Table 4). Each failure was reported by real system users and was caused by a configuration change (i.e., a value different from the default was used). These 64 failures have diverse root causes, including 51 misconfigurations and 13 software bugs exposed by *valid* configuration changes.[3] We collected failures from issue-tracking systems instead of user forums or mailing lists because: (1) failures recorded in issue-tracking databases tend to have had large impact, and (2) issue-tracking databases rigorously record the version of the systems on which the failures were reported, which is critical for reproducing failures. Importantly, we only generate ctests from the tests in the reported version, *not* from tests in later versions.

[3]For seven failures, misconfigurations triggered bugs in the code. We categorize them as "misconfigurations" in Table 4.

Software	Misconfigs	Bugs (Valid Configs)	Total
HCommon	11 (84.6%)	2 (15.4%)	13
HDFS	21 (95.5%)	1 (4.5%)	22
HBase	8 (61.5%)	5 (38.5%)	13
ZooKeeper	8 (66.7%)	4 (33.3%)	12
Alluxio	3 (75.0%)	1 (25.0%)	4
Total	51 (76.9%)	13 (20.3%)	64

Table 4: Statistics on real-world configuration-induced failures from issue-tracking databases used in ctest evaluation.

Root Cause	# Failures	# (%) Detected by Ctests	
		Gen Only	Gen + Rewrite
Misconfigurations	51	41 (80.4%)	51 (100.0%)
⊢ Corrupt config files	3	3 (100.0%)	3 (100.0%)
⊢ Value type errors	3	3 (100.0%)	3 (100.0%)
⊢ Out-of-range values	12	11 (91.7%)	12 (100.0%)
⊢ Value semantic errors	22	16 (72.7%)	22 (100.0%)
⊢ Dependency violations	10	7 (70.0%)	10 (100.0%)
⊢ Resource violations	1	1 (100.0%)	1 (100.0%)
Bugs exposed by valid config	13	10 (76.9%)	11 (84.6%)
Total	64	51 (79.7%)	62 (96.9%)

Table 5: Ctest effectiveness in detecting real-world configuration-induced failures of various root-cause types. Most types are self-explanatory; value semantic errors refer to misconfigurations that violate the semantics of the configuration parameter, including invalid file paths, URI, IP addresses, permission masks, etc.

Ctests evaluated. For each failure, we identify each configuration parameter p_i and its value v_i in the failure-inducing configuration change (13 of 64 failures involve more than one configuration parameter). We then generate ctests using the method in §4 for p_i. Further, we apply the two rewriting rules in §5.4 to enhance 11 generated ctests.

6.1.1 Effectiveness

Table 5 shows the effectiveness of ctests in detecting the 64 real-world failures and the root causes of those failures.

The results are promising. 96.9% (62/64) of the failure-inducing configurations are detected by ctests. *All* failures due to misconfigurations are detected. Specifically, 79.7% (51/64) of all failures are detected by using only generated ctests; the other 17.2% (11/64) require rewriting of ctests (§5.4). In 9 of the 11 failures that require rewriting, we only remove unnecessary value resets (like in Figure 6a). In the other two, we also change an assertion (like in Figure 6b). The results show that existing tests contain effective test logic and oracles needed to expose failure-inducing configuration changes. By leveraging those test logic/oracles, ctests can effectively detect failure-inducing configuration changes and prevent them from being deployed to production.

Failure Mode	Count (Pct)
Unexpected runtime exceptions	31 (50.0%)
Exceptions thrown by configuration-checking code	27 (43.5%)
Failing assertions in ctest code	3 (4.8%)
Test timeout (the system hangs)	1 (1.6%)

Table 6: Failure modes of ctests when detecting the failures.

By checking the behavior of code that exercise configuration parameters, ctests have generic ability to detect diverse types of misconfigurations, as well as bugs exposed by valid configuration changes (Table 5). That is, ctests are not designed to detect specific types of misconfigurations or bugs. We exemplified failures detected by ctests in Figures 1 and 2. Table 6 shows the failure modes of ctests on the 62 detected configuration-induced failures. Most failures manifested as unexpected runtime exceptions (division by zero, array index out of bound exceptions, etc.) or exceptions thrown by configuration-checking code. We show examples in Figures 2 and 7. Both types of exceptions would have the same impact on production systems if the failure-inducing changes were deployed. In three failures, test assertions fail because of unexpected behavior. The last failure was a test timeout that occurred because the configuration change caused the system to hang (similar to Figure 9a).

Two of the 64 failures were not detected by ctests [2,80]. In ALLUXIO-9810 [2], the root cause is a buggy shell script that no test invoked. The root cause of ZOOKEEPER-2299 [80] is a bug in a method that no test in the reported ZooKeeper version exercised. Both bugs can be detected by extending the test suite. In fact, for ZOOKEEPER-2299, the latest ZooKeeper version includes a test from which we have now generated a ctest that detects this bug.

6.1.2 Comparison with State-of-the-Art Techniques

Table 7 compares ctests with two state-of-the-art configuration checking techniques, PCheck [67] and Spellcheck [48]. Both PCheck and Spellcheck are designed for cloud systems and do not require additional training data or rule sets.

None of the 13 failures induced by *valid* configuration changes triggering bugs in code can be detected by existing configuration validation or automatic misconfiguration detection techniques, because those techniques only check whether configuration values are valid.

Ctests detected *all* misconfigurations among the real-world failures, including many that are challenging for state-of-the-art checking and detection techniques to detect. Spellcheck only detects value-type errors. In our real-world configuration-induced failure dataset (Table 5), only three failures were caused by value-type errors.

The following misconfigurations detected by ctests cannot be detected by PCheck: (1) two misconfigurations leading to non-crashing behavior (e.g., Figure 1), (2) five misconfigurations involving operations that have side effects (e.g., writing files), and (3) three misconfigurations that require client-side interactions to expose. Note that PCheck performs *post-deployment* configuration validation; PCheck does not run tests but instruments deployed systems. Ctests detect misconfigurations early, during *pre-deployment* testing. PCheck has two limitations that ctests do not have: (1) PCheck cannot have side effects in the production environment, and (2) PCheck cannot deal with external dependencies and events (e.g., client operations) [67]. Moreover, unlike PCheck, ctests can find bugs resulting from valid configuration changes.

	# Failures	Spellcheck	PCheck	Ctest	
				Gen Only	Gen+Rewrite
Misconfigs	51	3	41	41	51
Bugs	13	0	0	10	11
Total	64	3	41	51	62

Table 7: A comparison of Ctests, PCheck, and Spellcheck in detecting misconfigurations and bugs exposed by valid configuration changes (Table 5). We assume sound PCheck and Spellcheck static analyses—these are upper bounds for PCheck and Spellcheck.

6.2 Evaluating Ctests on Diverse Misconfigurations

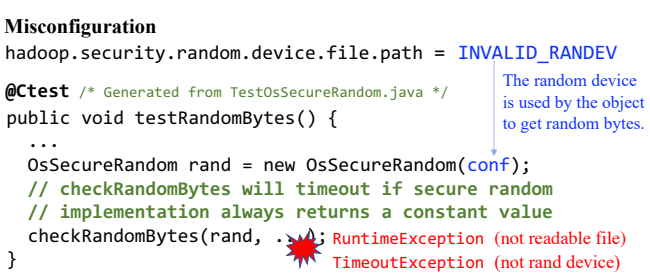
We ran ctests on injected misconfigurations to (1) systematically evaluate ctests' effectiveness on many diverse configuration parameters with different value types and semantics, and (2) experimentally evaluate ctests on misconfigurations that were not in the failures from issue-tracking databases.

Injected misconfigurations. We generate up to three erroneous values for each of the 392 configuration parameters in §5. We use the misconfiguration generation rules proposed for misconfiguration injection testing [31,32,69]. But we exclude rules such as case alternation and random fuzzing, which lead to many false errors. Note that the misconfiguration generation rules are different from the heuristics for generating *valid* values in §4.2.2. Specifically, we generate misconfigurations based on the types and semantics of each configuration parameter. For Boolean or enum types, we generate invalid options. For numeric types, we generate values containing alphabetic characters, and out-of-range values (smaller/larger than the min/max value). For parameters without explicit data ranges specified in the configuration file, we use the range of their data type, e.g., `INT_MAX` as the maximum value of integers. For strings, erroneous values are generated based on the semantics of the parameter. We follow the fine-grained rules defined in [32,69]. For example, for file-path parameters, we generate non-existent files, incorrect file content, and incorrect file types. We reviewed each generated erroneous value to reduce false errors.

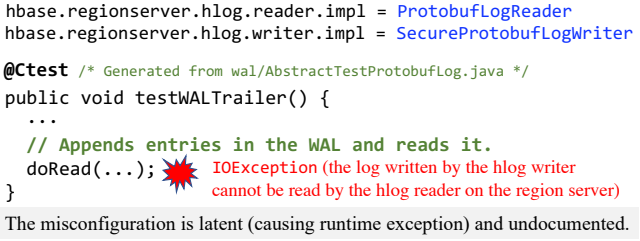
Ctests evaluated. We use the generated ctests from §5. For each erroneous value e generated for p, we create a configuration diff $D_e = \{(p \mapsto e)\}$ that sets p's value to e. We run all

Software	Complete	Partial	None	N/A
HCommon	43 (48.3%)	22 (24.7%)	24 (27.0%)	1
HDFS	67 (77.9%)	12 (14.0%)	7 (8.1%)	4
HBase	52 (61.9%)	23 (27.4%)	9 (10.7%)	6
ZooKeeper	20 (90.9%)	2 (9.1%)	0 (0.0%)	10
Alluxio	43 (47.8%)	15 (16.7%)	32 (35.6%)	0

Table 8: Ctest effectiveness in detecting injected misconfigurations *per parameter*. "Complete", "Partial", and "None" refer to number of parameters with all, some (but not all), and none of the injected misconfigurations detected, respectively. "N/A" refers to the number of parameters in which all the generated misconfigurations turned out to be valid due to the imprecision of error generation.



(a) Invalid file content. The ctest detects the misconfigurations by testing the functionality of the random device.



(b) Non-interoperability (undocumented [22]). The ctest detects the misconfigurations by testing the reader and writer together.

Figure 9: Non-trivial misconfigurations detected by ctests.

the ctests for p on each D_e and check whether any ctest fails on e. Unlike in §6.1, we do not rewrite ctests in this evaluation due to the larger size of experiments. So, our effectiveness results are a lower bound.

6.2.1 Effectiveness on Injected Misconfigurations

Table 8 shows the effectiveness of ctests in detecting the injected misconfigurations. Ctests detect *all* injected errors for 47.8%–90.9% of parameters and *at least one* injected error for 64.4%–100% of the parameters across the five systems.

Figure 9 shows two non-trivial misconfigurations detected by ctests. In Figure 9a, a ctest detects an invalid random device file path in HCommon by using the referred device to generate random bytes. Very few existing misconfiguration detection tools check file content; they mostly just check file paths and metadata. In Figure 9b, most `reader` and `writer` implementations of HBase are interoperable, but a few are not. Ctests checked the interoperability of a specific (`reader`, `writer`) pair and detected this non-trivial misconfiguration. The non-interoperability was neither documented nor checked in the system code before we reported it [22]. Using the non-interoperability configurations will fail HBase region servers.

Software	No Observable Symptom		Test Inadequacy	
	Correction	Mask	No Exposure	No Oracle
HCommon	14 (14.0%)	10 (10.0%)	56 (56.0%)	20 (20.0%)
HDFS	4 (11.8%)	8 (23.5 %)	9 (26.5%)	13 (38.2%)
HBase	25 (46.3%)	8 (14.8%)	19 (35.2%)	2 (3.7%)
ZooKeeper	2 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Alluxio	3 (2.7%)	0 (0.0%)	100 (90.9%)	7 (6.4%)
Total	48 (16.0%)	26 (8.7%)	184 (61.3%)	42 (14.0%)

Table 9: Root causes of false negatives among the injected misconfiguration *values*.

```
1 if (snapRetainCount < 3) {
2   LOG.warn(
3     "Invalid autopurge.snapRetainCount: "
4     + snapRetainCount + ". Defaulting to 3");
5   snapRetainCount = 3;
6 }
7 /* quorum/QuorumPeerConfig.java */
```

(a) An example of error correcting code in ZooKeeper

```
1 try { ...
2   paths = conf.get("dfs.datanode.shared.file.
        descriptor.paths")
3   fdFac = FileDescFactory.create(..., paths);
4   ...
5 } catch (IOException e) {
6   LOG.debug(
7       "Disabling ShortCircuitRegistry", e);
8 }
9 /* datanode/ShortCircuitRegistry.java */
```

(b) An example of partial-failure masking in HDFS

Figure 10: Two patterns that lead to false negatives during misconfiguration injection.

The generated ctests failed to detect 28.4% (300 of 1055) injected misconfigurations, i.e., false negatives. The results are consistent with the evaluation of misconfigurations without rewriting in §6.1. Recall that we do not rewrite tests in this evaluation, which could improve ctest adequacy (§6.1).

We inspected the 300 false negatives. Table 9 shows root causes of false negatives and their distribution. 75.3% of false negatives are due to inadequacy of ctests that either does not expose the effects of the misconfigurations or does not have oracles to check the effects. Many of these effects are non-functional (e.g., performance issues). Moreover, unlike real-world failures (§6.1), many injected misconfigurations are expected to be uncommon in practice. So, the systems have no error-checking logic or test code. For example, in HDFS, negative `io.seqfile.compress.blocksize` values cause se-

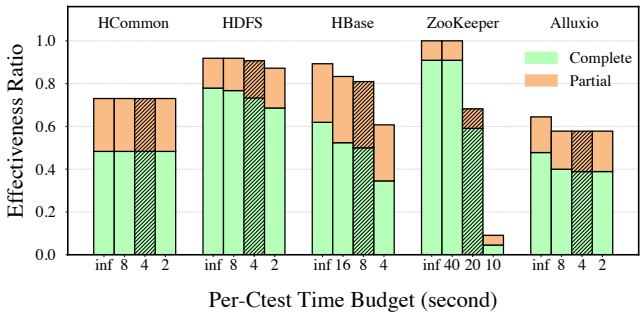


Figure 11: Time-budget analysis results. The results show the effectiveness of detecting misconfigurations if only ctests that finish under each time budget are run. The `inf` budget is equivalent to times from Table 8 where "Complete" and "Partial" are defined. We use the shaded budgets for experiments in §6.3.

vere performance degradation: every `append` triggers data compression. However, HDFS does not check against negative values nor have a test with performance-based oracles.

The remaining 24.7% of false negatives have no observable effects because of the presence of error-correcting code (e.g., Figure 10a), or because the consequences were masked (e.g., Figure 10b) by the system. Ctests cannot detect misconfigurations that have no observable effects.

6.2.2 Time-Budget Analysis

The per-parameter evaluation enables us to analyze the tradeoff between effectiveness and running time of ctests. To analyze this tradeoff, we performed a time-budget analysis. Our time-budget analysis excludes ctests that do not finish under a specified time budget and measures the effectiveness of the remaining ctests for detecting misconfigurations. We have not yet designed a test prioritization [45,72] scheme for ctests (§7), so we use per-test budgets (the amount of time *each* ctest is allowed to run) rather than a total-test-time budget (the amount of time *all* ctests are allowed to run). Per-test time budgets are well suited to test-suite parallelization, where each test is run in a separate process. Ctests for time-budget analysis run on an 8-core Intel i7-9700 CPU with 32 GB memory and Ubuntu 18.04.

Figure 11 shows the results of time-budget analysis. We observe that different budget ranges are needed for different systems given their different test characteristics. For example, ZooKeeper does not have many unit tests but relies mostly on integration tests. So, ZooKeeper needs larger per-test time budgets than other systems. Further, all ctests in HCommon finish under two seconds, so there is no decline in effectiveness across the time budgets shown. The key result from Figure 11 is that smaller per-test budgets can still achieve similar levels of effectiveness as running all the ctests for all projects except for Zookeeper. We use the shaded budgets in Figure 11 for evaluating ctests on real-world configuration files in §6.3.2 because they achieve good time-effectiveness tradeoff. We use a minimal per-test budget of 4 seconds to leave room for performance variability.

Software	# Files Tested	# Files that Fail Ctests		# False Alarms
		Version/Env.	Misconfig.	
HCommon	20	16	4	0
HDFS	20	15	2	0
HBase	20	12	0	0
ZooKeeper	20	14	0	0
Alluxio	12	3	1	0

Table 10: Results of running ctests on real-world configuration files collected from Docker images.

6.3 Evaluating Ctests on Configuration Files

We evaluate the effectiveness of ctests using configuration files collected from public Docker images. The experiments also enable us to measure the false positives and overhead of ctests on real-world configuration files (these are hard to systematically evaluate in §6.1 and §6.2).

Evaluated configuration files. We extract 92 configuration files from Docker images hosted on DockerHub [14] using the method described in [68]. We randomly sample 20 Docker images from the most popular 300 image repositories on DockerHub for the five systems. We only find 12 Alluxio image repositories that use non-default configuration files on DockerHub. We use the most recent image in each repository. The average number of configuration parameters in these files is 5.8 (the minimum is one and the maximum is 29).

Ctests evaluated. We generate ctests using the method in §4. For each configuration file f, we create a diff $D_f = \{(p \mapsto v_f)\}$ for all v_f explicitly set in f and run all the ctests that cover at least one parameter in D_f (see §3.2). We use the selected per-test budget from §6.2.2 to run ctests on each configuration file. We run ctests against the configuration files on our server, rather than deploying the ctest infrastructure in each image's container to reduce the cost of resolving dependencies and setting up environments (many images are built from old OS distributions with incompatible dependencies).

6.3.1 Ctests Effectiveness on Configuration Files

Table 10 presents the effectiveness of ctests on real-world configuration files. Surprisingly, many configuration files did not pass the ctests. We inspected all failed ctests and found 85 of 537 values to be erroneous. 76 of 85 erroneous values are correct in the container, but fail ctests because (1) certain files, IP addresses and ports in the containers do not exist or are not available on our server, and (2) the ctests are generated from tests from a newer version of the system—the configuration values in the images are no longer correct. We reported one such case, ALLUXIO-3402 [1], where the configuration parameter `alluxio.user.file.metadata.load.type` has the value "*Always*" in an image, `scality/alluxio`. But, in the latest Alluxio, an all-capitalized parameter value is required.

Software	Ctests with Budget		# All Ctests		
	# Ctests	Runtime	# Ctests	Runtime	Baseline
HCommon	1014.20	1.90	1019.55	3.42	3.84
HDFS	1850.75	37.48	2310.20	126.35	120.43
HBase	842.75	47.70	1053.65	99.47	140.86
ZooKeeper	39.55	6.79	76.95	26.29	19.98
Alluxio	796.5	1.66	796.5	1.66	1.44

Table 11: The number of ctests and their running time (in minutes) per configuration file using all ctests and ctests within time budget (selected in §6.2.2). The numbers are averaged over all evaluated files. "Baseline" is the time for running the corresponding original tests (not ctests).

So a ctest generated for the latest Alluxio fails. These results show that ctests effectively detect misconfigurations caused by version and environment changes [76].

Ctests also detect 9 misconfigurations of various types in seven configuration files (Table 10), including malformed files, value-type errors, and dependency violations, which are misconfigurations in the native container. Based on our inspection on the seven Docker images, we suspect that some of these configuration files may be managed by custom scripts that overwrite those files. Unfortunately, we find no documentation for five of the seven Docker images on DockerHub.

Zero false positives found. We expected a few false positives due to tests that assume some values but were not identified when generating parameter sets for ctests—the heuristics for generating valid values for validation are unsound (§4.2.2). However, we found no false positives (Table 10).

6.3.2 Ctest Running Time on Configuration Files

We measure the ctest-running time per configuration file. Table 11 shows the average total ctest-running time per configuration file when running ctests that finish within the per-test time budgets selected in §6.2.2. HCommon, ZooKeeper and Alluxio take less than ten minutes. HDFS and HBase have longer-running tests and take few tens of minutes.

We run all ctests with the `inf` budget and compare it with running the ctests under the time budget. There is no difference in the effectiveness of ctests, showing that the budgets are sufficient. We also compare total running time of all ctests with a baseline total time for running all the original software tests from which the ctests are generated. The results show that the running times of the ctests are similar to those of the original software tests ("Baseline" in Table 11). The running time for HBase is about 70% of its baseline because many tests in HBase aborted the execution and failed quickly due to the exceptions triggered by misconfigurations.

7 Discussion and Limitations

There is no silver bullet against configuration-induced failures. Ctests offer a simple, effective way to detect failure-inducing configurations, and are complementary to existing techniques.

The effectiveness of generated ctests depends on the adequacy of the original tests. On the evaluated systems, which have abundant tests, ctests outperform state-of-the-art tools. However, ctests cannot be generated if there are no existing tests, which is why no ctest exposed the two bugs in the evaluation (§6.1.1). Mature software systems will likely benefit from ctests because they have comprehensive test suites [29, 51]. For newer projects or projects with less comprehensive test suites, the generation of ctests could be limited. Note that the concept of ctests is not limited by the generation method discussed in §4. Ctests can be implemented by developers, just like they implement regular software tests.

Ctests cannot localize the root causes of configuration-induced failures. Based on our analysis of ctest results (§6.2 and §6.3), root cause localization can usually be done efficiently by analyzing the stack traces. However, a few failures take considerable time to understand, due to (1) complexity of configuration value propagation and transformation, or (2) unexpected, hidden configuration constraints (e.g., Figure 8). Fault localization [64] for configuration-induced failures can be developed to automate root cause analysis.

Ctests can increase the cost of regression testing, which is already expensive. Section 6.3.2 shows that running ctests for the evaluated systems takes reasonable time. On the other hand, we believe that the cost of running ctests can be significantly reduced by developing ctest reduction, prioritization and minimization techniques, as was done for regression testing [72]. One direction is to analyze ctest code and to understand how each ctest exercises configuration changes, towards reducing and prioritizing ctests. Ctests running time can also be further reduced by running ctests in parallel.

The ctest generation described in §4 is neither sound nor complete. First, the heuristics for detecting implicit test assumptions (§4.2.2) are unsound and could lead to false negatives in detecting bugs. Our heuristics minimize false positives. Second, dynamically tracing test executions to identify parameters exercised in tests (§4.2.1) is incomplete, because configuration changes could lead to different execution paths. Like any other form of testing, we do not claim completeness.

Like any other pre-deployment testing, ctests are fundamentally limited by a possible mismatch between the test environment and the production environment. Such a mismatch could lead to both false positives and false negatives.

8 Related Work

The severity and prevalence of configuration-induced failures [18, 19, 34, 35, 40, 42, 55, 74] has resulted in novel techniques for misconfiguration troubleshooting and debugging [3, 4, 46, 61–63, 73]. Advanced techniques have also been developed for diagnosing production failures [10, 13, 30, 79]. Ctests proactively detect failure-inducing configuration changes to *prevent* production failures in the first place.

Ctests is complementary to our prior work, PCheck [67]. We compared ctests with PCheck [67] in §6.1.2, despite

PCheck being a *post-deployment* technique. Note that PCheck can only detect misconfigurations, because it considers only statements on the data-flow path of each configuration value. Differently, ctests can detect valid configuration changes that expose bugs in the code, a common type of failure-inducing configuration changes [55]. Techniques designed for pre- and post-deployment have fundamentally different opportunities and challenges. In our experience, it is difficult (if not impossible) for auto-generated checking code to deal with many sophisticated real-world misconfigurations. This was the main motivation behind ctests which can exercise code and configurations together. But post-deployment techniques such as PCheck do not have problems caused by the mismatches between the test and the production environments.

We mentioned in §3.2 that ctests are complementary to configuration validation and misconfiguration detection [6, 27, 38, 43, 44, 49, 50, 55, 59, 61, 62, 66, 67, 75, 77], similar to how software testing complements static bug detection tools. Ctests can detect failure-inducing configuration changes that are challenging for existing techniques to detect, e.g., valid configuration changes that expose bugs. Most automated detection techniques only focus on specific types of misconfigurations. For example, Rex [38] detects dependency violations between source-code files and configuration files which should be updated together. Ctests are not specific to any type of misconfigurations or softwre bugs—they detect failure-inducing configuration changes based on the resulting program behavior. A common class of existing validation/detection techniques requires validation rules or training data that either do not exist (e.g., for systems that we evaluate) or are not available (we found no rule sets or training data online). In contrast, ctests do not rely on external rule sets or training data—they leverage existing abundant test cases.

Ctests complement software and system testing. In essence, ctests enhance existing testing techniques to focus on the actual configurations in production or configurations to be deployed, given that testing all possible configurations is infeasible. A ctest is a parameterized test. But ctests differ from traditional parameterized unit tests (PUTs) [57, 58] in goal, parameter source, and generation method. The goal of PUTs is to allow developers rerun the same test against different inputs, to cover more program paths. The goal of ctests is to connect production system configurations to software tests, to find failure-inducing configuration changes. The inputs to PUTs are either specified by developers or automatically generated by symbolic execution, but the inputs to ctests are read from the system configuration files or diffs.

9 Conclusion

This paper proposes ctests to connect software testing with production system configurations to enable detecting failure-inducing configurations during testing. We present how to generate ctests from existing software tests that are abundant in mature cloud systems. We show that ctests are effective in detecting real-world failure-inducing configurations, including both misconfigurations and dormant software bugs exposed by valid configuration changes. Our goal of ctests is to make testing of configuration changes a key component of configuration management and fill the missing piece in the practice of treating configuration as code. We have made all the code and datasets available at: https://github.com/xlab-uiuc/openctest.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Haryadi Gunawi, for their insightful comments and feedback. We thank Darko Marinov, Lalith Suresh, Neil Zhao, Madhusudan Parthasarathy, Yongle Zhang, David Chou, and Justin Meza for the invaluable discussions. We thank Qingrong Chen, Andrew Yoo, Angello Astorga, Liia Butler, and Jonathan Osei-Owusu for helping proofread the paper. This work was funded in part by CCF-1816615, CCF-2029049, CNF-1956007, CCF-2019277, a Facebook Distributed Systems Research award, Microsoft Azure credits, and Google Cloud credits.

References

[1] ALLUXIO-3402. Backward compatibility for enum-typed configuration. https://alluxio.atlassian.net/browse/ALLUXIO-3402, 2020.

[2] ALLUXIO GITHUB ISSUE #9810. Alluxio worker fails to start when using multiple storage media in single tier on EMR. https://github.com/Alluxio/alluxio/issues/9810, 2019.

[3] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (October 2012).

[4] ATTARIYAN, M., AND FLINN, J. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (October 2010).

[5] BARROSO, L. A., HÖLZLE, U., AND RANGANATHAN, P. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2018.

[6] BASET, S., SUNEJA, S., BILA, N., TUNCER, O., AND ISCI, C. Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware'17), Industrial Track* (December 2017).

[7] Bazel: a fast, scalable, multi-language and extensible build system. https://bazel.build/, 2020.

[8] BELL, J., LEGUNSEN, O., HILTON, M., ELOUSSI, L., YUNG, T., AND MARINOV, D. DeFlaker: Automatically Detecting Flaky Tests. In *In Proceedings of the 40th International Conference on Software Engineering (ICSE'18)* (May 2018).

[9] BEYER, B., MURPHY, N. R., RENSIN, D. K., KAWAHARA, K., AND THORNE, S. *Site Reliability Workbook: Practical Ways to Implement SRE*. O'Reilly Media Inc., August 2018.

[10] BHAGWAN, R., KUMAR, R., MADDILA, C. S., AND PHILIP, A. A. Orca: Differential Bug Localization in Large-Scale Services. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (October 2018).

[11] Buck: A fast build tool. https://buck.build/, 2020.

[12] CHEN, Q., WANG, T., LEGUNSEN, O., LI, S., AND XU, T. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *In Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (November 2020).

[13] CUI, W., GE, X., KASIKCI, B., NIU, B., SHARMA, U., WANG, R., AND YUN, I. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (October 2018).

[14] Docker Hub. https://www.docker.com/products/docker-hub, 2020.

[15] FOWLER, M. Eradicating Non-Determinism in Tests. https://martinfowler.com/articles/nonDeterminism.html, April 2011.

[16] Gradle Build Tool. https://gradle.org/, 2020.

[17] GRAVES, T. L., HARROLD, M. J., KIM, J.-M., PORTER, A., AND ROTHERMEL, G. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology 10*, 2 (April 2001), 184–208.

[18] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (November 2014).

[19] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)* (October 2016).

[20] HADOOP-10508. RefreshCallQueue fails when authorization is enabled. https://issues.apache.org/jira/browse/HADOOP-10508, 2014.

[21] HBASE-22559. [RPC] set guard against CALL_QUEUE_HANDLER_FACTOR_CONF_KEY. https://issues.apache.org/jira/browse/HBASE-22559, 2019.

[22] HBASE-23962. Improving the documentation for 'hbase.regionserver.hlog.reader, writer.impl'. https://issues.apache.org/jira/browse/HBASE-23962, 2020.

[23] HDFS-15124. Crashing bugs in NameNode when using a valid configuration for 'dfs.namenode.audit.loggers'. https://issues.apache.org/jira/browse/HDFS-15124, 2020.

[24] HDFS-15250. Setting 'dfs.client.use.datanode.hostname' to true can crash the system because of unhandled UnresolvedAddressException. https://issues.apache.org/jira/browse/HDFS-15250, 2020.

[25] HDFS-7684. The host:port settings of the daemons should be trimmed before use. https://issues.apache.org/jira/browse/HDFS-7684, 2015.

[26] HDFS-7727. Check and verify the auto-fence settings to prevent failures of auto-failover. https://issues.apache.org/jira/browse/HDFS-7727, 2015.

[27] HUANG, P., BOLOSKY, W. J., SIGH, A., AND ZHOU, Y. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the 10th ACM European Conference in Computer Systems (EuroSys'15)* (April 2015).

[28] HUO, C., AND CLAUSE, J. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)* (November 2014).

[29] IVANKOVIĆ, M., PETROVIĆ, G., JUST, R., AND FRASER, G. Code Coverage at Google. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)* (August 2019).

[30] KASIKCI, B., SCHUBERT, B., PEREIRA, C., POKAM, G., AND CANDEA, G. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures. In *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)* (October 2015).

[31] KELLER, L., UPADHYAYA, P., AND CANDEA, G. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)* (June 2008).

[32] LI, S., LI, W., LIAO, X., PENG, S., ZHOU, S., JIA, Z., AND WANG, T. ConfVD: System Reactions Analysis and Evaluation Through Misconfiguration Injection. *IEEE Transactions on Reliability 67*, 4 (December 2018), 1393–1405.

[33] LILLACK, M., KÄSTNER, C., AND BODDEN, E. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering (TSE) 44*, 12 (December 2018), 1269–1291.

[34] LIU, H., LU, S., MUSUVATHI, M., AND NATH, S. What bugs cause production cloud incidents? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'19)* (May 2019).

[35] MAURER, B. Fail at Scale: Reliability in the Face of Rapid Change. *Communications of the ACM 58*, 11 (November 2015), 44–49.

[36] Apache Maven. http://maven.apache.org/, 2020.

[37] MEDEIROS, F., KÄSTNER, C., RIBEIRO, M., GHEYI, R., AND APEL, S. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)* (May 2016).

[38] MEHTA, S., BHAGWAN, R., KUMAR, R., ASHOK, B., BANSAL, C., MADDILA, C., BIRD, C., ASTHANA, S., AND KUMAR, A. Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (February 2020).

[39] MUKELABAI, M., NEŠIĆ, D., MARO, S., BERGER, T., AND STEGHÖFER, J.-P. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)* (September 2018).

[40] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (December 2004).

[41] NARLA, C., AND SALAS, D. Hermetic Servers. https://testing.googleblog.com/2012/10/hermetic-servers.html, October 2012. Google Testing Blog.

[42] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)* (March 2003).

[43] PALATIN, N., LEIZAROWITZ, A., SCHUSTER, A., AND WOLFF, R. Mining for Misconfigured Machines in Grid Systems. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)* (August 2006).

[44] POTHARAJU, R., CHAN, J., HU, L., NITA-ROTARU, C., WANG, M., ZHANG, L., AND JAIN, N. ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'15)* (August 2015).

[45] QU, X. Configuration Aware Prioritization Techniques in Regression Testing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)* (May 2009).

[46] RABKIN, A., AND KATZ, R. Precomputing Possible Configuration Error Diagnosis. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)* (November 2011).

[47] RABKIN, A., AND KATZ, R. Static Extraction of Program Configuration Options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)* (May 2011).

[48] RABKIN, A. S. *Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software*. PhD thesis, University of California, Berkeley, 2012.

[49] SANTOLUCITO, M., ZHAI, E., DHODAPKAR, R., SHIM, A., AND PISKAC, R. Synthesizing Configuration File Specifications with Association Rule Learning. In *Proceedings of 2017 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)* (October 2017).

[50] SANTOLUCITO, M., ZHAI, E., AND PISKAC, R. Probabilistic Automated Language Learning for Configuration Files. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)* (July 2016).

[51] SAVOIA, A. Code coverage goal: 80% and no less! https://testing.googleblog.com/2010/07/code-coverage-goal-80-and-no-less.html, July 2010. Google Testing Blog.

[52] SAVOR, T., DOUGLAS, M., GENTILI, M., WILLIAMS, L., BECK, K., AND STUMM, M. Continuous Deployment at Facebook and OANDA. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)* (May 2016).

[53] SHERMAN, A., LISIECKI, P., BERKHEIMER, A., AND WEIN, J. ACMS: Akamai Configuration Management System. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)* (May 2005).

[54] SHIEBER, J. Facebook blames a server configuration change for yesterday's outage. https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage/, March 2019.

[55] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)* (October 2015).

[56] THE APACHE HBASE REFERENCE GUIDE. Default Configuration. https://hbase.apache.org/book.html#hbase_default_configurations, 2020.

[57] TILLMANN, N., AND SCHULTE, W. Parameterized Unit Tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)* (September 2005).

[58] TILLMANN, N., AND SCHULTE, W. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE Software 23*, 4 (July 2006), 38–47.

[59] TUNCER, O., BILA, N., ISCI, C., AND COSKUN, A. K. ConfEx: An Analytics Framework for Text-Based Software Configurations in the Cloud. Tech. Rep. RC25675 (WAT1803-107), IBM Research, March 2018.

[60] WACKER, M. Just Say No to More End-to-End Tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html, April 2015. Google Testing Blog.

[61] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (December 2004).

[62] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)* (October 2003).

[63] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (December 2004).

[64] WONG, W. E., GAO, R., LI, Y., ABREU, R., AND WOTAWA, F. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering (TSE) 42*, 8 (August 2016), 707–740.

[65] XIANG, C., HUANG, H., YOO, A., ZHOU, Y., AND PASUPATHY, S. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)* (July 2020).

[66] XIANG, C., WU, Y., SHEN, B., SHEN, M., HUANG, H., XU, T., ZHOU, Y., MOORE, C., JIN, X., AND SHENG, T. Towards Continuous Access Control Validation and Forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)* (November 2019).

[67] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (November 2016).

[68] XU, T., AND MARINOV, D. Mining Container Image Repositories for Software Configurations and Beyond. In *In Proceedings of the 40th International Conference on Software Engineering (ICSE'18), New Ideas and Emerging Results* (May 2018).

[69] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating System Principles (SOSP'13)* (November 2013).

[70] XU, T., AND ZHOU, Y. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR) 47*, 4 (July 2015).

[71] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (October 2011).

[72] YOO, S., AND HARMAN, M. Regression Testing Minimisation, Selection and Prioritization: A Survey. *Software Testing, Verification, and Reliability 22*, 2 (March 2012), 67–120.

[73] YUAN, C., LAO, N., WEN, J.-R., LI, J., ZHANG, Z., WANG, Y.-M., AND MA, W.-Y. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys'06)* (April 2006).

[74] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (October 2014).

[75] YUAN, D., XIE, Y., PANIGRAHY, R., YANG, J., VERBOWSKI, C., AND KUMAR, A. Context-based Online Configuration Error Detection. In *Proceedings of 2011 USENIX Annual Technical Conference (USENIX ATC'11)* (June 2011).

[76] ZHANG, G., AND LIU, L. Why Do Migrations Fail and What Can We Do about It? In *Proceedings of the 25th USENIX Large Installation System Administration Conference (LISA'11)* (December 2011).

[77] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)* (March 2014).

[78] ZHANG, S., AND ERNST, M. D. Which Configuration Option Should I Change? In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (May 2014).

[79] ZHANG, Y., RODRIGUES, K., LUO, Y., STUMM, M., AND YUAN, D. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)* (October 2019).

[80] ZOOKEEPER-2299. NullPointerException in LocalPeerBean for ClientAddress. https://issues.apache.org/jira/browse/ZOOKEEPER-2299, 2015.

[81] ZOOKEEPER-3721. PR #1266: ZOOKEEPER-3721: Making the boolean configuration parameters consistent. https://github.com/apache/zookeeper/pull/1266, 2020.

Providing SLOs for Resource-Harvesting VMs in Cloud Platforms

Pradeep Ambati[‡] Íñigo Goiri[‡] Felipe Frujeri[‡] Alper Gun[†]
Ke Wang[†] Brian Dolan[†] Brian Corell[†] Sekhar Pasupuleti[†]
Thomas Moscibroda[†] Sameh Elnikety[‡] Marcus Fontoura[†] Ricardo Bianchini[‡] *

[†]Microsoft Azure [‡]Microsoft Research

Abstract

Cloud providers rent the resources they do not allocate as evictable virtual machines (VMs), like spot instances. In this paper, we first characterize the unallocated resources in Microsoft Azure, and show that they are plenty but may vary widely over time and across servers. Based on the characterization, we propose a new class of VM, called Harvest VM, to harvest and monetize the unallocated resources. A Harvest VM is more flexible and efficient than a spot instance, because it grows and shrinks according to the amount of unallocated resources at its underlying server; it is only evicted/killed when the provider needs its minimum set of resources. Next, we create models that predict the availability of the unallocated resources for Harvest VM deployments. Based on these predictions, we provide Service Level Objectives (SLOs) for the survival rate (*e.g.*, 65% of the Harvest VMs will survive more than a week) and the average number of cores that can be harvested. Our short-term predictions have an average error under 2% and less than 6% for longer terms. We also extend a popular cluster scheduling framework to leverage the harvested resources. Using our SLOs and framework, we can offset the rare evictions with extra harvested cores and achieve the same computational power as regular-priority VMs, but at 91% lower cost. Finally, we outline lessons and results from running Harvest VMs and our framework in production.

1 Introduction

Motivation. Cloud providers usually rent their resources to customers as Infrastructure as a Service (IaaS) VMs. When deployed, each VM consumes a fixed amount of resources from the server where it lands. Customers can keep their VMs for seconds or years [16] and may request more VMs over time. Thus, providers need to provide the illusion of perfectly elastic resources (*e.g.*, by reserving demand growth buffers) while operating the infrastructure with high availability (*e.g.*, by transparently handling hardware failures). For these reasons, they need to leave unallocated capacity.

To monetize this unallocated capacity, providers offer VMs with relaxed SLOs at discounted prices. Specifically, they offer low-priority evictable VMs, often called spot VMs [1, 8, 14]. These VMs are evicted if their resources are needed by regular-priority (or simply regular) on-demand VMs. Thus, evictable VMs are ideal for customers to run batch jobs or other workloads that can tolerate evictions, at very low cost.

Unfortunately, an evictable VM cannot consume all the unallocated resources of a server unless it fits perfectly in it. Even if it does, a large evictable VM will be promptly evicted whenever even a single resource is needed by a newly arriving regular VM. Multiple small evictable VMs can allocate the same amount of resources but will add overhead to operate more VMs. In addition, their larger number of evictions introduce VM re-creation and application re-initialization overheads that may even cause unavailability.

Given these limitations of existing evictable VMs, we argue that there should be a new class of evictable VMs able to *dynamically* and *flexibly* harvest all the unallocated resources of any server on which they land.

Our work. We first characterize the unallocated resources in Microsoft Azure. The characterization shows that there is potential for harvesting these resources, but they fluctuate over time and their availability is heterogeneous across servers and clusters. The characterization unearths the dynamics of the unallocated resources over multiple time durations.

Next, we propose a new class of evictable VM, called Harvest VM, as a novel way to monetize unallocated resources. A Harvest VM has a minimum size in terms of its physical resources, but it dynamically receives more or fewer physical resources beyond this minimum, depending on the amount of unallocated resources at its underlying server. A Harvest VM is only evicted if its minimum size is needed for a regular VM. In this paper, we focus on harvesting CPU cores.

Provisioning applications to run on harvested resources is challenging. However, we can predict the availability and amount of the unallocated resources in the datacenter. We use these predictions to provide SLOs for Harvest VM deployments. The SLO specifies the probability for a Harvest VM to survive for a certain period and how many resources it will get on average. For example, if a customer wants to create 100 VMs, the SLO may indicate that 90% of them

*Ambati is affiliated with the Univ. of Massachusetts Amherst, but was at Microsoft Research during this work. Gun and Wang are now with Google.

will survive for more than 1 day, with an average of 10 cores. The provider does not monitor or actively seek to meet each individual SLO; instead, we retrain our prediction models frequently and provide our SLO as a statistical estimate [12]. As such, our SLOs can be considered predictions or estimates over large numbers of Harvest VMs, rather than guarantees.

Renting unallocated resources is cheap, but requires applications to manage the evictions. In addition, with Harvest VMs, the amount of resources backing each VM can vary. Harvest VMs are most useful when the applications they run can adapt to the number of available resources. For example, many applications use thread pools and can naturally adapt their parallelism. Others can schedule more load on larger VMs. The provider can hide these complexities by using Harvest VMs to create cheap SaaS (Software-as-a-Service), PaaS (Platform-as-a-Service), and FaaS (Function-as-a-Service) offerings. In fact, Harvest VMs are ideal for cluster scheduling (*e.g.*, Apache YARN [37], Kubernetes [22]) and serverless (*e.g.*, AWS Lambda [32], Azure Functions [4]) frameworks. These frameworks can schedule more tasks/functions on a Harvest VM that has grown to use more physical cores, and stop scheduling tasks/functions on one that has lost physical cores. To demonstrate how to adapt these frameworks, we build Harvest Hadoop to schedule computation (*e.g.*, data-processing, machine learning training) on harvested resources.

Our evaluation shows that we accurately predict the unallocated resources and provide SLOs. We predict the survival rate of a VM for 1 hour with an average error under 2% and lower than 6% for longer terms. We also predict the additional cores that can be harvested within a fraction of a core on average. Our SLOs and framework allow us to run Hadoop workloads on Harvest VMs at 91% lower cost to the customer than regular VMs, by offsetting the rare evictions with additional harvested cores. Compared to standard evictable VMs, the cost savings can reach 47%. Finally, we discuss lessons and results from deploying Harvest VMs and Harvest Hadoop in production to run internal workloads in Azure.

Summary. Our contributions are:

- We characterize the unallocated resources of a large cloud.
- We propose Harvest VMs to harvest unallocated resources.
- We build predictors for the availability of unallocated resources and provide a new SLO for these resources.
- We build Harvest Hadoop, a cluster scheduling framework to leverage Harvest VMs.
- We discuss lessons and results from our production deployment of Harvest VMs and Harvest Hadoop.

2 Background and related work

Deploying VMs. Each VM deployment targets a geographical region, which is partitioned into clusters of servers that have the same hardware. Each region may have a different number of clusters and hardware mix. A region-level scheduler decides which VMs go to which clusters based on several factors (*e.g.*, hardware required, maintenance tasks, available capacity) [19]. These factors can cause clusters to have different VM loads, even in the same region. Then, a cluster-level scheduler decides which server in the cluster will run each VM. When a VM is assigned to a server, a server-level agent creates the VM and manages its lifecycle.

Evictable VMs. Providers sell their excess capacity at discounted prices as evictable VMs [1, 8, 14]. These VMs are evicted/killed when the provider needs the capacity (*e.g.*, due to a spike in the number of on-demand VMs). Providers notify the VMs before they evict them: GCP and Azure provide a 30-second warning, whereas AWS gives 2 minutes.

Variable-resource VMs. Sharma *et al.* [33] recently proposed Deflatable VMs, which change *virtual* resources dynamically (via hot-plugging/unplugging), and a multi-level resource reclamation approach for explicitly adapting applications, operating systems, and hypervisors to the available resources. They also combined reclamation with deflation-aware VM scheduling. We believe that expecting the whole stack to adapt is unrealistic in practice. Instead, we favor simplicity and maintainability for *production deployment*: (1) we minimize the changes to the cloud platform, so deploying Harvest VMs is no different than deploying any other VM, and the VM scheduler is unaware that Harvest VMs grow and shrink; (2) we do not change the number of virtual cores, and instead transparently vary the number of physical cores.

A more aggressive VM design could harvest the unallocated cores *and* any allocated cores that are temporarily idle. This is out of the scope of this paper. Instead, we focus on the usability of core-harvesting VMs (aggressive or otherwise) in practice with SLOs and software for them. Our SLOs can be extended for aggressive harvesting, whereas Harvest Hadoop can be used directly.

Like a Harvest VM, a burstable VM [7, 13] has a fixed number of virtual cores and receives a minimum number of physical cores. However, it is only allowed to burst (*i.e.*, receive additional physical cores) up to its maximum size, after accumulating enough "credits" by staying below a predefined core utilization. A Harvest VM differs in that (1) it harvests as many cores as are unallocated for as long as they remain so, *i.e.* there is no concept of credit; and (2) it is evictable. These characteristics mean that providing SLOs for Harvest VMs is also quite different than for burstable VMs.

Resource harvesting. Other approaches to resource harvesting have either focused on running batch workloads on idle machines (*e.g.* [25, 26]) or co-locating batch workloads with latency-sensitive services on bare-metal servers (*e.g.* [23, 27, 38, 39, 46, 47]). In contrast, we focus on a virtualized infrastructure where physical resources are reserved for the VMs that allocate them (as is the norm in the public cloud), and predict the availability and dynamics of the unallocated resources to produce SLOs.

Characterization and SLOs. To indirectly characterize the unallocated resources at cloud providers, prior work [2, 9, 31,

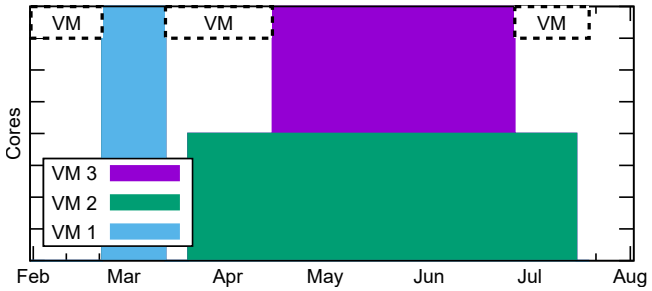


Figure 1: Allocation of VMs on a server, including hypothetical VMs (dashed) that consume unallocated resources.

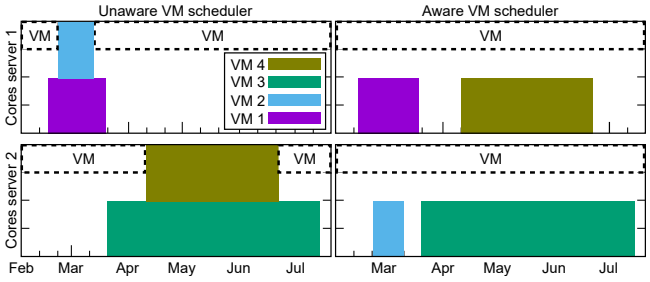


Figure 2: VM allocations on two servers in our characterization (left) and when the VM scheduler is aware of VMs consuming unallocated capacity (right).

34] has analyzed publicly available traces of EC2 spot prices. Using the traces, they tried to model the availability of spot instances. In contrast, we use actual resource allocation data from the entire Azure server fleet to characterize the resources more accurately and comprehensively.

From the perspective of the provider, Carvalho *et al.* [12] characterized the reclaimable resources in 6 Google clusters. They aggregated the cluster-wide resources and predicted their availability for long-term (6-month) SLOs. They did not consider VM evictions or how the reclaimable resources vary at each server. However, the majority of VMs live less than 1 day and get deployed in relatively small groups [16]. Hence, we quantify the unallocated resources *per server* at a *fine time granularity*. Moreover, our SLOs quantify VM survival rates and average numbers of cores over horizons as short as 1 hour.

3 Characterizing unallocated resources

In this section, we characterize the potential for resource harvesting and the dynamics of the unallocated capacity in Azure. The characterization is affected by the Azure VM scheduler [19]. However, the scheduler behaves similarly to those of other providers [38] by tightly packing VMs while ensuring that it can find big enough holes for large VMs.

Methodology. We analyze the resource allocation in Azure from February to October 2019. The data we present does not include confidential metrics, such as number of servers or percentage of unallocated resources. However, the trends we illustrate are enough for the purposes of this paper.

We compute the allocated resources in each server based on the regular VMs running over time, *i.e.* we exclude resources that have been allocated to existing evictable VMs. We account for the main resources (*i.e.*, cores, memory, storage, and network bandwidth) for both the VMs and the servers. We then check if we could allocate in each server a hypothetical evictable VM of a minimum size, for how long, and how many unallocated resources it could potentially get.

In more than 80% of cases where we could not allocate the hypothetical VM, the scarcest resource (*i.e.*, the one that prevents the allocation) is cores. This is not surprising as Azure matches its hardware and VM sizes to have a single dominant resource and simplify capacity management. In the vast majority of remaining cases, disk space is the constraint. Thus, if we can find unallocated cores at a server, the other resources will most likely be unallocated as well.

Figure 1 shows an example server that runs 3 VMs over six months with the allocation of cores on the Y-axis. In early February, there are no VMs allocated to the server so we can run a 1-core hypothetical VM (dashed box) during that time. In late February, *VM 1* starts and takes the full server so we cannot run any other VM. Once *VM 1* finishes, the server becomes empty so we can run another hypothetical VM. *VM 2* starts in late March but it only takes half of the server, so we can keep running the hypothetical VM until *VM3* starts. In this period, we could place 3 hypothetical VMs with an average lifetime of almost one month.

This figure shows the hypothetical VMs with a fixed size but there are plenty of additional unallocated cores still left in the server. For example, when the hypothetical VM can run, at least half of the cores are unallocated.

Our characterization is *pessimistic* in that the unallocated resources are actually *more stable* in practice. For example, our characterization may find the scenario on the left side of Figure 2, which shows two servers with real VMs and hypothetical VMs. However, if the VM scheduler were to actually allocate VMs to consume the unallocated resources, it could allocate the real VMs differently to avoid evicting the hypothetical VMs as on the right side of the figure.

Temporal patterns. A key aspect to quantify is how long we could run a hypothetical evictable VM to consume unallocated resources in each server. Figure 3 shows how many servers could host a 1-core VM with 16GB of memory and 200GB of disk for a given time (*e.g.*, 1 hour, 1 day) in a popular region. We do not list the actual numbers of servers on the Y-axis for confidentiality reasons. Considering 1 hour into the future, we can see a daily pattern where there are more unallocated resources at night. For 1 day, we can see a weekly pattern and how weekends have substantially more unallocated resources. Once we consider the next week, the temporal pattern is not as clear. Overall, the longer horizon numbers show a decrease in unallocated resources over time. These data show that it is important to account for the time of day and day of the week (at least implicitly) when predicting the unallocated resources, especially for shorter periods.

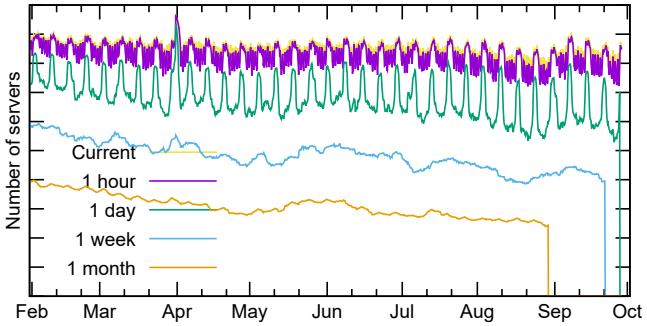


Figure 3: #servers with 1 unallocated core in a region.

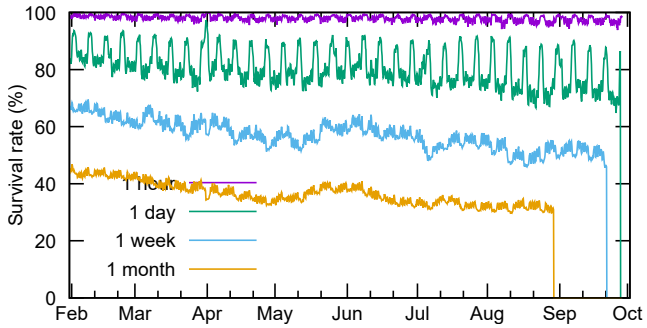


Figure 4: Survival rate with 1 unallocated core in a region.

From these numbers, we can compute the survival rate, *i.e.* the percentage of these evictable VMs that would survive for a given time (*e.g.*, dividing the "1-hour" values by the corresponding "Current" values computes the percentage of VMs that would survive for 1 hour). Figure 4 shows this survival rate over time. For example, it shows that in April, an average of roughly 60% of the 1-core evictable VMs (at most one per server) would survive for one week.

Cluster behaviors. As we discuss in Section 2, clusters may behave differently even within a region. Figure 5 shows how many servers could host a 1-core evictable VM in one specific cluster in the same region as Figure 3. In both late May and early June, the number of allocated VMs increased substantially, each time leaving less unallocated capacity. This shows that the amount of unallocated resources can change drastically over time. There are multiple reasons for such an effect, but in this case it was due to a shift in load across clusters, driven by the higher level across-clusters scheduler. These results show that we must consider each cluster individually when predicting the available unallocated resources.

Aggregating across all regions. So far, we have discussed servers in 1 region. Now, we discuss aggregate data over all regions. First, we consider the average durations over which at least 1 core is unallocated at each server. Over all regions, most servers can host a 1-core evictable VM for at least 1 hour on average. This number drops by 40% for 1 day and by another 40% for 1 month. As expected, fewer servers have at least 1 unallocated core for long periods (*e.g.*, 1 month) than short ones (*e.g.*, 1 hour). Moreover, even when servers have the same overall amount of unallocated capacity over time

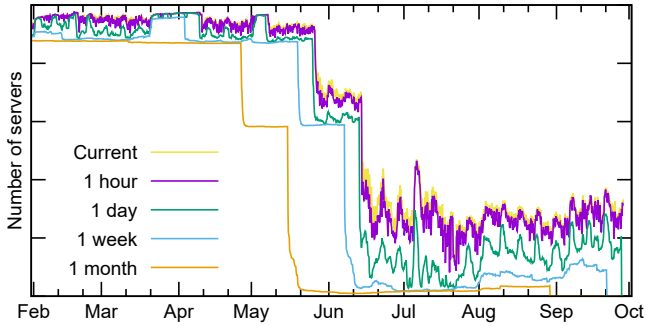


Figure 5: #servers with 1 unallocated core in a cluster.

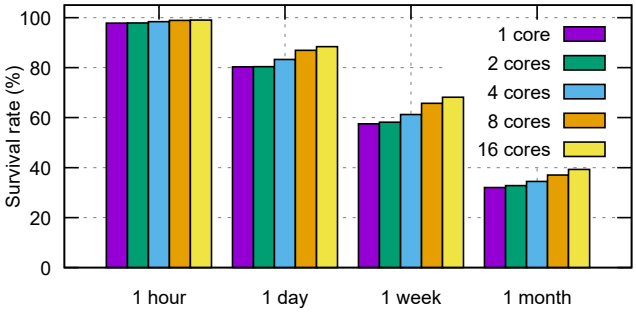


Figure 6: Survival rate of deployable evictable VMs as a function of lifetime and minimum size.

(measured in core×hours), they may be able to host widely different numbers of evictable VMs: servers that tend to have short periods with unallocated cores can host many (short-lived) evictable VMs, whereas those that tend to have long periods with unallocated cores host fewer (long-lived) VMs.

Next, we consider the average survival rate of the deployable 1-core VMs (at most one per server), again aggregating across all regions. The purple bars in Figure 6 plot the average survival rate for all deployable 1-core VMs for 1 hour, 1 day, 1 week, and 1 month. These four bars compute the average of the curves in Figure 4 but for all regions. Almost 100% of the VMs would survive for 1 hour, but only 80% of them would survive for of 1 day and 32% would survive for 1 month.

Minimum unallocated cores. These results quantify the survival rate of 1-core evictable VMs. However, many servers have more unallocated cores than 1. For example, only 55% of the servers have 4 unallocated cores (*i.e.*, capable of hosting a 4-core evictable VM) for at least 1 hour on average.

Figure 6 also plots the average survival rate of other minimum sizes (at most one VM per server). Larger deployed evictable VMs tend to survive longer than smaller ones, even though they are less likely to find a server where to run. For example, 88% of the 16-core VMs survive for 1 day or longer, but only 80% of the 2-core VMs survive for that long. This effect is due to the cluster-level scheduler trying to pack new VMs tightly in servers that are already closer to being full.

Additional unallocated cores. The results above consider evictable VMs that consume a minimum number of unallocated cores. However, as shown in Figure 1, there are many

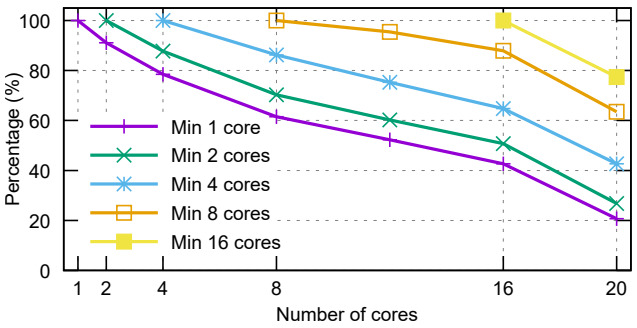


Figure 7: Percentage of deployable evictable VMs that could have received more cores, for each minimum size.

periods when there are additional unallocated resources in the server. Figure 7 shows the percentage of evictable VMs of each minimum size that could potentially have been as large as 1, 2, 4, 8, 16, and 20 cores. For example, 78% of 1-core VMs could have gotten 4 or more cores, and 85% of the 4-core evictable VMs could have gotten 8 or more cores. These results illustrate that (1) a large percentage of the (more numerous) small VMs could have been much larger; and (2) a large percentage of the (less numerous) large VMs could have been even larger. However, allocating a larger evictable VM on a server increases the chance that it will be evicted when the additional cores are needed for higher priority VMs.

Another important consideration is how stable the set of additional cores is, *i.e.* how quickly the set changes due to core allocations/deallocations. We find that 94% of these state changes last for more than 1 second, 90% of them last for more than 5 seconds, 50% of them last for more than 10 minutes, and 10% of them last more than 3 hours. Clearly, the set of additional cores is stable enough that they could be effectively harvested and used by applications.

Multiple VMs per server. So far, we have discussed deploying at most one evictable VM in each server. However, the results above show that there are often enough unallocated resources for more VMs and the amount of these resources varies over time. Under these conditions, the provider can maximize the amount of unallocated resources it monetizes via evictable VMs with as many 1-core VMs as will fit in each server at each point in time. Unfortunately, a larger number of VMs per server increases management (more evictions) and resource (more copies of the guest OS) overheads. The key problem is that *standard evictable VMs are not the ideal abstraction to maximize the use of unallocated resources while keeping overheads down.*

High-level takeaways. Our characterization shows that:

1. There are many unallocated resources that can be harvested. However, they fluctuate significantly over time. There are plenty of unallocated resources for a short time but many fewer for longer periods.
2. These resources are not evenly distributed across clusters. A cluster's allocation may also change drastically over time.
3. The available unallocated resources vary substantially depending on amount (minimum size) and duration. Smaller minimum sizes are more widely available but they do not survive as long as larger minimum sizes.
4. There are many additional unallocated resources in each server beyond this minimum size that can be harvested. The additional resources vary over time at a fairly coarse granularity, but trying to harvest them with standard evictable VMs could cause many evictions and waste resources.

4 Harvest Virtual Machines

Section 3 shows that there are plenty of unallocated resources that can be harvested, while takeaway #4 suggests that doing so with standard evictable VMs is not ideal. Thus, we propose a new class of evictable VM, called *Harvest VM*, that dynamically grows and shrinks to harvest as many unallocated resources as available on the server where it runs. With Harvest VMs, we maximize the resource harvesting at each server, while keeping evictions and overheads down.

Overview. Users select a minimum size for each Harvest VM. A Harvest VM starts with as many unallocated physical resources as are available in its host server, but grows and shrinks dynamically after that. For example, a Harvest VM may have 4 physical cores as its minimum size. At server selection time, this VM is assigned to a server that has at least 4 unallocated cores. Say this server has 20 cores. At creation time, the Harvest VM would be created with 20 *virtual* cores and would receive an initial number of *physical* cores equal to the number of unallocated cores in the server (at least 4 cores, of course). During its lifetime, the Harvest VM will grow (*i.e.*, receive more physical resources) when a co-located regular VM terminates and shrink (*i.e.*, lose physical resources) when a new regular VM lands on the same server. Since the Harvest VM changes size only when other VMs arrive/terminate, these changes occur fairly infrequently (Section 3). As a Harvest VM has lower priority, it is evicted/killed if the cloud platform needs its minimum size for a regular VM.

As an example, Figure 8 shows a server with 8 physical cores that hosts 2 regular VMs with 2 cores each. At t_0, a Harvest VM with a minimum size of 2 cores lands on the server. As there are unallocated cores, the Harvest VM grows to 4 cores. At t_1, VM 2 finishes and the Harvest VM grows to 6 cores. At t_2, VM 3 with 4 cores lands on the server and the Harvest VM shrinks to 2 cores (its minimum size). At t_4, VM 4 with 2 cores lands on the server, causing the Harvest VM to be evicted as it would have to shrink below its minimum size.

Production implementation in Azure. We create a new family of hyperthreaded Harvest VMs that users can select from. The family defines VM types with a minimum size of 1, 2, or 4 cores (*i.e.*, 2, 4, and 8 hyperthreads, respectively). The smallest Harvest VM has a minimum of 1 core, 16GB of memory, 200GB of disk with 3k IOPS, and 1Gbps of network bandwidth. The resources for the larger sizes scale proportionally to the number of minimum physical cores.

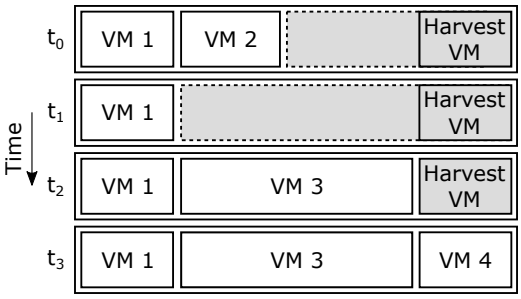


Figure 8: Harvest VM dynamically changing sizes over time.

Our current implementation only harvests physical cores; the other resources stay fixed during the Harvest VMs' lifetimes. The Harvest VMs can grow to use all physical cores of the server, as this fits nicely our current production uses (Section 6), which can consume as many cores as are available. For simplicity, the implementation does not allow more than one Harvest VM per server. We discuss upcoming changes to this design in Section 8.

Users can deploy many Harvest VMs to a region at the same time. The provider deploys the Harvest VMs in the same way that it deploys any other VM. Ultimately, each Harvest VM is scheduled onto a server by a cluster-level VM scheduler. On each server, Azure runs the Hyper-V hypervisor [29] and an agent responsible for managing VMs locally, including VM creation, termination, and physical core reassignment across VMs. The agent uses hypercalls for assigning a Harvest VM to a group of cores and capping the amount of CPU time the group receives. To prevent cache interference between a Harvest VM and the co-located regular VMs, the agent constrains the Harvest VM to a subset of cache ways of the last-level cache, using cache allocation technology [15].

The changes in the number of physical cores are not directly visible by the Harvest VM, as its number of virtual cores does not change. However, the application or scheduling framework running on a Harvest VM may want to take advantage of any harvested cores. Thus, we expose the number of currently assigned physical cores to the Harvest VM via the KVP mechanism of Hyper-V [30]. Applications or frameworks can use this information to adapt their behaviors. For example, a scheduling framework can assign more tasks to a Harvest VM that has just received more cores.

The scheduler may evict a Harvest VM (1) when it needs the minimum resources for a regular VM, or (2) proactively to avoid the eviction latency when it expects that its minimum resources will be needed soon. In either case, the scheduler informs the Harvest VM about the upcoming eviction, and gives it 30 seconds to shutdown cleanly. At deployment time, users can specify whether they want another Harvest VM to be created (on a different server) to replace an evicted one.

Comparison to standard evictable VMs. Unlike evictable (*e.g.*, spot) VMs, Harvest VMs are only evicted when the provider needs their minimum resources for higher priority VMs. In addition, Harvest VMs harvest additional unallocated cores from the servers that host them. In Figure 8, using evictable VMs to harvest those additional cores would have caused them all to be killed at t_2, whereas the Harvest VM shrinks and avoids the high eviction overhead. Due to the additional harvested cores, it takes many more evictable VMs to harvest as many cores as Harvest VMs, implying higher management and resource overheads. In Section 7.5, we show that evictable VMs also imply higher costs to users.

Using Harvest VMs. Harvest VMs are most useful when workloads can gracefully adapt to evictions and a time-varying number of physical cores. First, workloads must be able to continue operating correctly after VM evictions. An eviction is similar to a server failure, so all practical distributed applications are already capable of handling them. Embarrassingly parallel applications handle these failures even more easily. Regardless of application type, users often want new (evictable) VMs to be created to replace evicted VMs, and cloud platforms already provide this functionality. However, as VM re-creation and application re-configuration are expensive, users can make informed decisions about their Harvest VM deployments using our SLOs.

Second, applications must be able to leverage additional cores and degrade gracefully when cores are removed. To do so, applications can check the number of currently assigned physical cores and adapt accordingly. Core re-assignments are much cheaper than VM re-creation and re-configuration, so applications can more easily handle them. For example, the application may create (destroy) software threads when more (fewer) cores are available or have a thread pool where work can wait for cores. Despite their lower overhead, users can use our SLOs to know how many cores to expect per Harvest VM, so they can provision enough threads and VMs.

Still, providers may decide that Harvest VMs are not ideal as an IaaS offering. Instead, they can use them to implement cheaper SaaS, PaaS, or FaaS offerings. In fact, our current Azure deployment uses Harvest VMs to implement a core-harvesting version of Hadoop.

Privacy/confidentiality. On individual servers, Harvest VMs reveal the VM arrival and departure events. However, they do not threaten the confidentiality of the cloud platform's resource utilization, as long as determined (and well-funded) users are not allowed to deploy Harvest VMs to most servers. To avoid this, the provider can simply establish an overall quota of Harvest VMs in each region. The privacy of the workloads is also protected, as Harvest VMs do not reveal any info about (1) co-located regular VMs to the users of Harvest VMs, or (2) their workloads to the provider or co-located regular VM users. In addition, using Harvest VMs for SaaS, PaaS, or FaaS adds an extra software layer that further reduces the chance of leaking sensitive information.

Pricing and deployment cost. A detailed pricing discussion is beyond our scope. Instead, we assume that users pay (in $\$/(core \times hours)$) the same for their Harvest VM minimum size as a standard evictable VM of equal size (evictable VMs

are already heavily discounted compared to regular VMs), and get a further discount on any additional cores beyond the minimum (billing for these cores can be per-use or per-allocation to the Harvest VM). This pricing scheme is beneficial for both users, who can rent resources cheaply, and the provider, who can aggressively monetize its unallocated capacity.

To compute the cost to the user of a deployment of multiple Harvest VMs, we need to consider evictions. An eviction forces the re-creation of the VM at another server, which takes the time to instantiate the VM and restore the application. This results in a loss in useful compute power (measured in $core \times hours$). Thus, the average cost per useful core hour (in $\$/(core \times hours)$) of a deployment is:

$$\frac{minsize\ core\ hrs \times price + additional\ core\ hrs \times \alpha \times price}{minsize\ core\ hrs + additional\ core\ hrs - recovery\ core\ hrs}$$

where *minsize core hrs* is the total core hours for the VMs' minimum size, *additional core hrs* is the total number of cores hours harvested beyond the minimum size, α is the extra discount the provider offers on the additional cores ($\alpha = 0$ means those cores are free and $\alpha = 1$ means they cost the same as the minimum size cores), and *recovery core hrs* is the total amount of core hours spent recovering from evictions.

Harvesting other unallocated resources. Our current implementation only harvests cores. Many workloads can use additional cores (*e.g.*, ML training and most data analytics) with stable needs for other resources. Yet, harvesting other resources would make Harvest VMs more broadly beneficial, so we are building prototypes for harvesting some of them.

Harvesting network and disk bandwidth are similar to core harvesting (they are all compressible resources). Current hypervisors manage bandwidth limits and set them up when starting each VM. To harvest these resources, the server agent can dynamically change the limits. For applications or frameworks to be aware of changes, we expose these values to the Harvest VM using our existing mechanisms.

Harvesting memory is more challenging. Current hypervisors support dynamically changing the memory assigned to a VM. When adding new memory, this shows as hot-plugged memory in the VM. When removing memory, the guest OS uses memory ballooning to make some part of it unavailable. This may trigger swapping in the Harvest VM and the applications/frameworks should be aware. If the VM cannot free up memory, the operation may crash (or ungracefully evict) the VM. Other works discuss similar approaches [33].

For disk space, VMs usually mount a virtual disk (VHD) for data. A naive option would be to extend and shrink the VHD. Extending a VHD can be done while it is mounted, but shrinking it requires unmounting and compressing. Another option would be to add and remove full VHDs depending on the disk space available in the server. Both approaches are intrusive and require applications/frameworks to be aware.

5 Providing SLOs for Harvest VMs

Our characterization showed that the amount of unallocated resources to run Harvest VMs varies over time, in terms of temporal patterns (*e.g.*, daily and weekly) and across-cluster behavior changes (*e.g.*, shift in load across clusters). Moreover, the VM scheduling dynamics produce numerous smaller sets of unallocated resources that survive shorter times, and fewer larger sets that survive longer. These factors make it difficult for users to provision the right minimum size and number of Harvest VMs.

To ease this task, we predict the survival rate of the Harvest VMs and the amount of resources they are likely to receive on average, and provide these predictions to users in the form of an SLO. The SLO is a best-effort statistical estimate as in prior work [12], so the provider should retrain the prediction models frequently (*e.g.*, every day). The provider need not monitor or actively try to enforce each SLO individually, which would be impractical. Nevertheless, the SLO enables applications beyond just batch workloads to use Harvest VMs, as long as they can tolerate the occasional eviction and the core reassignments (Section 6).

Our predictions leverage machine learning (ML) models and features we can collect in production.

User input and SLO definition. The user must first inform her desired number of Harvest VMs (*e.g.*, 100), minimum size (*e.g.*, 2 physical cores), and region. Based on these requirements, we provide an SLO for the survival rate and the number of additional cores for a set of predefined time horizons: 1 hour, 1 day, 1 week, and 1 month. For example, the survival rate SLO for each horizon can be: 60% of the Harvest VMs will likely survive at least 1 hour, 40% will likely survive at least 1 day, 25% will likely survive at least 1 week, and 15% will likely survive at least 1 month. We also provide confidence intervals (*e.g.*, between 55% and 70% will last 1 hour with 95% confidence).

For each horizon, our SLO also estimates the average number of additional cores. For example, the Harvest VMs will likely receive an average of 5-7 cores with 95% confidence for the first hour, 8-11 cores over the first day, etc.

If the SLO is not acceptable, users can change the number and/or minimum size of the Harvest VMs they request. If no SLO is acceptable after multiple tries, users may opt for a mix of regular and Harvest VMs or select a different region. Once the Harvest VMs are running, users can check for updated SLOs, which become more accurate over time. Based on this updated information, they can adapt their deployments.

ML models and features. To provide the SLO, we use ML models to predict the survival rates and average sizes for each time horizon. After experimenting with multiple modeling approaches, we settled on Random Forest regressors [10].

The features we use in our models are as follows.

Cluster characteristics: This includes (1) number of servers, (2) number of racks, (3) generation of the hardware (including

their sizes), and (4) total resources (*e.g.*, cores and memory) in the cluster. Clusters with similar characteristics (*e.g.*, same type of servers) are likely to have similar behaviors. This is useful for new clusters without much historical data or clusters that have not seen particular conditions (*e.g.*, high allocation).
Cluster name: The identifier for the cluster helps improve the prediction accuracy for a specific cluster. This complements the cluster characteristics and still allows learning from the historical data from similar clusters.
Total resources allocated: This includes the total number of cores and memory (*e.g.*, in GBs) currently allocated to regular VMs in the cluster. Together with the cluster characteristics, we can compute the allocation percentage.
Number of VMs: This is the total number of VMs currently running in the cluster. The ratio between resources allocated and the number of VMs gives insights on how large the VMs in the cluster are. This is particularly useful to estimate the sets of unallocated resources in the cluster for Harvest VMs.
Auto Regressive: These are previous time series values of the outputs we want to forecast. This feature is especially useful because, as our characterization shows, past values are a reasonable indicator of the current values. Each output will use values for different past periods. For example, if we are predicting the survival rate for Harvest VMs in 1 day, this would include the evictions we actually saw in the last day.
Moving Average: This is similar to the Auto Regressive feature, but it smooths the past values using averages. We use multiple periods for the averages (*e.g.*, 1 hour, 1 day). This feature is useful to filter out peaks and reduce noise.
ML training and inference. We can *train* our models using data from Harvest VMs that ran in production in the past. This data includes the aspects that we want to predict (*e.g.*, how long the VM lasted for), the characteristics of the Harvest VM (*e.g.*, a minimum of 2 cores), and the state of the cluster at each point in time. However, as Harvest VMs have not run in production long enough, we use traces from production as our training data in this paper (Section 7.1).

At model *inference* time (*i.e.*, an SLO needs to be shown to a user), we first check which cluster in the desired region would potentially host the Harvest VMs that are being requested, and use the cluster characteristics and name for the cluster as input features for the inference. If the Harvest VM deployment is to be split across multiple clusters, we then predict for each one independently.
Discarded features. Other features' impact on prediction quality was small or even detrimental. Some of them are:
Number of VMs of each type: A VM type defines the number of cores, memory size, if it has GPUs, etc. There are hundreds of types and the model cannot make sense of them. Some features we use (*e.g.*, total number of VMs and cores allocated) are proxies and enable our models to infer this data concisely.
Date/time: These features were used in [36]. We do not include them, as features like the total number of VMs already carry implicit temporal patterns (*e.g.*, weekdays vs weekends).

Predicting standard evictable VM survivability. Our survival rate predictions can be directly applied to standard evictable VMs. In fact, we are working on a simpler version of our model to provide survival rate predictions for evictable VMs in production. The uses for these predictions are similar to the ones for Harvest VMs.

6 Harvest Hadoop

Cluster scheduling frameworks, such as Apache YARN [37] or Kubernetes [22], are good targets for Harvest VMs. A large number of applications already run on them, and they can be adapted to use Harvest VMs transparently to applications. These frameworks are built to handle server/VM failures, so they can be easily extended to manage evictions. Applications built for these frameworks, like Spark [44], also manage the straggler tasks that might result from an eviction. Moreover, these frameworks can be modified to schedule more tasks on a Harvest VM that has grown to use more cores, and stop scheduling new tasks on a Harvest VM that has lost cores. To demonstrate how to adapt these frameworks, we build *Harvest Hadoop* to schedule computation on harvested resources.
Harvest Hadoop architecture. Harvest Hadoop is an extension to the Hadoop [3] ecosystem. Hadoop includes the YARN cluster scheduler [37], which enables running many applications (*e.g.*, Spark [44], Flink [11]) to leverage harvested resources. It also includes the Hadoop Distributed File System (HDFS), which is optimized for large data files.

A key goal for Harvest Hadoop was to minimize the number of intrusive changes to YARN and HDFS, so that our system would be simple and practical, and our changes could be more easily contributed to open-source Hadoop. With this in mind, we design Harvest Hadoop with the following main features:
- It executes the YARN and HDFS master processes (called Resource Manager and Name Node, respectively) on regular VMs, as it is expensive to manage the failure of the masters;
- It executes the YARN and HDFS worker processes (called Node Manager and Data Node, respectively) on Harvest VMs;
- It uses storage within each Harvest VM simply as a cache of remote data (from the provider's highly available storage service), as evictions do not leave enough time for fully decommissioning a storage server; and
- It introduces a Harvest VM Manager (HVM Manager) that monitors the number of resources currently available to its Harvest VM and the informs the master processes. The master processes act accordingly at the next heartbeat.

We have contributed all the needed code changes to open-source Hadoop 3.3.0 [40,41]. Figure 9 illustrates the architecture, showing a server that has two regular VMs and a Harvest VM consuming all the resources not used by the regular VMs.
Managing evictions. We leverage the decommissioning feature in YARN [42]. When the provider notifies the Harvest VM that it will be evicted, the HVM Manager notifies the YARN Resource Manager (RM) to kill the containers in that

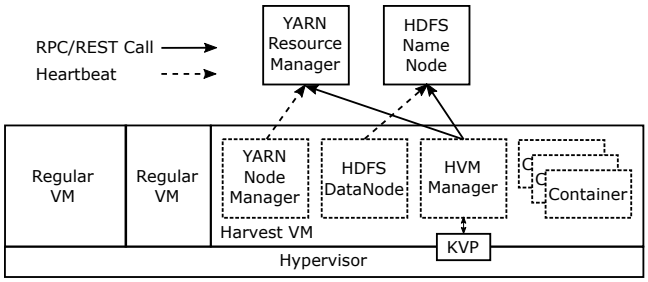


Figure 9: Architecture of Harvest Hadoop.

VM. If the worker gets evicted before killing all the containers, the RM will handle this as a failure and re-schedule the containers. Otherwise, the applications can decide whether they want to re-schedule any of their killed containers.

Evictions are more intrusive for data storage. We leverage the decommissioning mechanisms available in HDFS [20]. The HVM manager catches the eviction notification from the provider and tells the HDFS Name Node to start decommissioning the corresponding Data Node. As the advance notice to evict a VM is usually short (*e.g.*, 30 seconds), there is little time to decommission a full data node. This is the reason why we only cache remote replicas in Harvest VMs, as we mention above. HDFS replicates the cached files in other Harvest VMs, and uses a write-back policy for them [21].

At Harvest Hadoop deployment time, the user specifies (as an auto-scaler option) whether she wants her Harvest VM deployment to be replenished by the cloud platform to its original number of Harvest VMs when an eviction occurs.

Managing core reassignments. To adapt the scheduling, we leverage the resource updates in the existing heartbeats to the YARN RM. Zhang *et al.* took a similar approach in the bare-metal scenario [47]. The HVM manager periodically checks the number of cores assigned to its Harvest VM, and it notifies the RM if the number has changed.

If the Harvest VM gets more cores, the RM can now assign more containers to the VM. If the VM shrinks, the scheduler can: (1) kill some containers and let the application handle it as a failure, (2) run the containers in a deprived mode and wait until the application terminates them, or (3) notify the application to free up some containers.

Our current implementation uses a combination of the three options. The RM first selects the containers that should be killed based on their priorities and whether they are opportunistic [43]. Then, it notifies the applications in case they can terminate the containers. After a grace period (30 seconds), if the cores are still not enough, it will terminate the containers. This period allows graceful termination and can correct for the number of cores increasing again.

Harvesting other resources. We also modify Hadoop to be aware of the VMs' memory allocation, so it will work out-of-the-box when Harvest VMs become capable of harvesting memory. When the Harvest VM gets more memory, Harvest Hadoop can just deploy more containers to it. However, when the Harvest VM shrinks, we cannot run in deprived mode, unless the VM allows swapping to disk. For this reason, we keep a buffer of unused unallocated memory. The HVM manager notifies the Harvest VM when memory from this buffer is allocated, so that it can free up some of its own memory. The HVM manager kills the Harvest VM if the buffer is exhausted, *i.e.* the Harvest VM cannot release memory fast enough.

Hadoop does not need changes to benefit from harvested network and disk bandwidth, as applications can automatically use any additional bandwidth that becomes available.

7 Evaluation

7.1 Methodology

Our evaluation focuses on two sets of results. First, we assess the benefits of Harvest VMs and the accuracy of our SLOs. Ideally, we would do this assessment based on real production data. However, our SLOs are not in production yet. Moreover, the set of conditions under which we can evaluate our SLOs is limited with the production deployments of Harvest VMs. For these reasons, we use a *validated* simulator and production VM data for 25 clusters for our SLO evaluation.

Second, we explore the *real implementation of Harvest VMs and Harvest Hadoop in the provider's production infrastructure*. We use two large clusters: one running internal production VM workloads, and another running VM workloads that stress the cluster.

Simulator. We use Azure's own cluster simulator, which executes the real VM scheduler [19] code in assigning VMs to physical servers. Thus, the simulator closely mimics the constraints and preferences in the real scheduler. We feed the simulator with production VM arrival traces, and add Harvest VMs continuously to fill the cluster (*i.e.*, no new Harvest VMs can be allocated). We run the simulations in real Harvest VMs, *i.e.* each harvested core allows us to run one more (single-threaded) simulation in parallel.

We validate the simulator by comparing the number of Harvest VMs that can be created in a real cluster (based on logs of VM assignments to servers) and in simulation (replaying the corresponding VM arrival trace). Figure 10 shows this validation for Harvest VMs of 3 minimum sizes over 1 week, where dashed lines represent the real executions and solid lines represent simulated executions. The curves match closely because the simulator mimics the packing per server accurately. We also validate using longer periods and other clusters, and find an absolute average error of just 3%.

As inputs for our simulations, we use VM arrival data from 25 randomly selected clusters across 14 regions, including relatively small satellite regions. The data was collected from December 1st 2019 to March 1st 2020. We process the VM arrival data to generate the allocation state of each server every 10 minutes. The clusters exhibit a wide range of behaviors in terms of how highly allocated they are on average and how

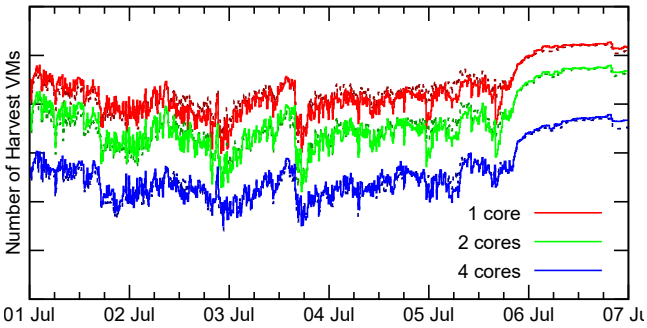


Figure 10: Simulation validation over 1 week. Solid curves are actual data; dashed curves are simulated.

stable the total allocated resources are over time. The number of servers per cluster ranges from several hundred to several thousand. There is no correlation between cluster size and allocation percentage or stability. When training our models, we use data from December 1^{st} 2019 to January 15^{th} 2020. We use the other 45 days for evaluating predictions.

Real experiments. For our real experiments, we use the Harvest Hadoop implementation we describe in Section 6. We configure the provider's deployment system to replace any Harvest VMs that get evicted, so that the overall number of Harvest VMs stays fixed during our experiments.

We use two large clusters, which we call *private* and *canary*. The private cluster has over 1700 servers and runs VMs that implement a production key-value store. The VM load is fairly stable over time. We create Harvest VMs in this cluster and attach them to a Harvest-Hadoop-based production data analytics and ML training system. We have deployed Harvest VMs to other private clusters as well, but selected this one for our results because it has been the most extensively used.

The canary cluster has around 650 servers and runs a synthetic VM load that stresses the provider's production infrastructure. This cluster is in the top percentile in terms of VM creations and terminations, and produces many resource allocation changes and evictions. For our experiments with this cluster, we create full Hadoop clusters. Each cluster consists of 3 Name Nodes and 2 Resource Managers (which run on regular VMs) and Harvest VMs that we scale on demand. For coordination, we also deploy a 5-node ZooKeeper 3.6.0 stamp in the same regular VMs. We run synthetic jobs, including MapReduce (*e.g.*, TeraGen and TeraSort) and Spark.

7.2 Benefits of Harvest VMs

We start the evaluation by assessing the benefits of Harvest VMs over standard evictable VMs in terms of numbers of VMs and evictions. Our comparison simulates the 25 production clusters in two scenarios: one in which we consume all the clusters' unallocated resources using evictable VMs, and another where we consume them using Harvest VMs. We place as many evictable 1-core VMs as will fit; larger sizes would not consume many unallocated resources. In the Harvest VMs scenario, 1 core is the minimum size and we only place one Harvest VM per server. In both scenarios, each VM has 16GB of memory and 200GB of disk.

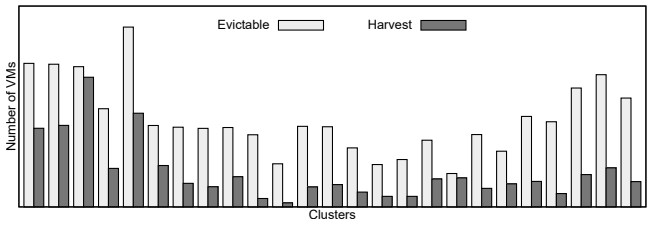


Figure 11: Number of VMs required to consume the unallocated resources of 25 production clusters.

Figure 11 shows the number of VMs required to consume the unallocared resources with evictable and Harvest VMs for each cluster. Across all clusters, we need between 8% and 10.7× (3.7× on average) more evictable VMs than Harvest VMs. The number of evictions of evictable VMs is also much higher by ∼ 3.6× on average. These results quantify our earlier observation that standard evictable VMs incur higher management and resource overheads than Harvest VMs.

7.3 Accuracy of SLOs for Harvest VMs

The accuracy of our SLOs hinges on our ability to accurately predict survival rates and average numbers of harvested cores. We start our evaluation with a detailed analysis of prediction accuracy for a few sample clusters, and then offer a global view of all clusters. The last part of the section evaluates our ML model and studies its sensitivity to multiple parameters.

Detailed analysis. Let us first consider the accuracy of our survival rate SLOs. For a cluster with fairly stable load, the graphs on the left of Figure 12 show the number of Harvest VMs with a minimum size of 4 cores that can be created (top), those that would survive 1 day (middle), and their survival rate after 1 day (bottom) over time. Each graph shows the actual and predicted values (with 95% confidence intervals), as well as the corresponding absolute errors. We plot predictions and errors every 10 minutes, given the actual cluster state at those times. For example, if the actual value is 100 and the prediction is between 90 and 120 with 95% confidence, the absolute error is 0%. Instead, if the actual value is 60, the error is -33% (*i.e.*, (60-90)/90) and the absolute error is 33%. The vertical line at January 15^{th} marks the split between the training and test datasets. The graph on the right shows the CDF of the errors comparing the actual survival rates to our predictions with and without 95% confidence intervals, during the test period. These would be the error distributions of our 1-day survival rate SLO.

The top graph shows that our predictions for the number of VMs that can be created are very accurate, even though the training data is almost a flat line and there are substantial variations after January 15^{th}. Our model recognizes that these behaviors are unknown and leverages the data from other

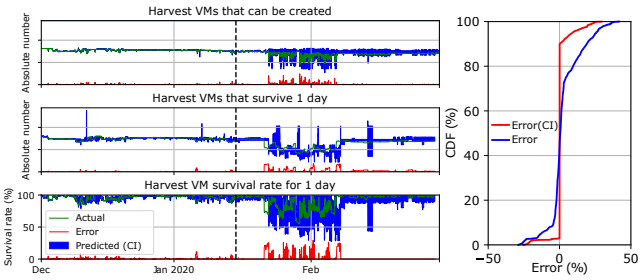


Figure 12: Predictions for Harvest VMs with 4-core minimum size and their survival rates after 1 day for a stable cluster.

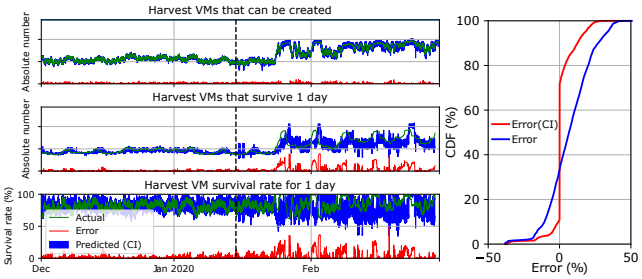


Figure 13: Predictions for Harvest VMs with 4-core minimum size and their survival rates after 1 day for the worst cluster.

clusters to provide a prediction that is not as precise. So, when those variations occur, the confidence intervals widen. The middle graph also shows very good accuracy when predicting the number of VMs that would survive after 1 day. Most importantly, the bottom graph and the CDFs to the right show that 80% of the absolute errors are very close to 0%, and 95% of them are lower than 15%, when compared to the predictions with confidence intervals. Errors are larger when comparing to exact predictions, but 90% of them are still lower than 20%. These results show very good accuracy for our SLO.

For comparison, we now study the cluster with the largest 99th-percentile errors in our dataset in Figure 13. Again, we assume 4-core Harvest VMs and 1 day survival. The top graph shows very low absolute errors, despite the significant change in behavior after January 15th. In contrast, the middle and bottom graphs show significant errors at times, despite the wider confidence intervals. The CDFs show the distribution of the errors in our SLO prediction. In this case, 85% of the predictions are within 10% and 95% within 20%. Thus, even in the worst cluster, our SLO would still provide valuable guidelines (*e.g.*, an actual survival rate of 85% while we predicted 65%).

We now study the accuracy of our predictions of average number of harvested cores. Figure 14 shows these predictions for Harvest VMs with 1 (top), 2 (middle), and 4 (bottom) minimum cores in another cluster with significant unallocated capacity. The horizontal lines show the minimum and maximum numbers of cores for each VM. The figure shows that a Harvest VM with a minimum size of 1 core gets between 6 and 14 cores in the cluster. During the test phase, the prediction accuracy is very good, showing that our average cores SLO would be accurate for this cluster. In addition, we can see

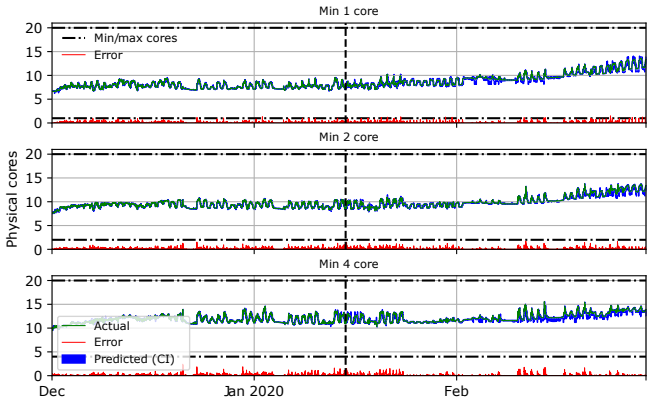


Figure 14: Prediction of unallocated cores in one cluster.

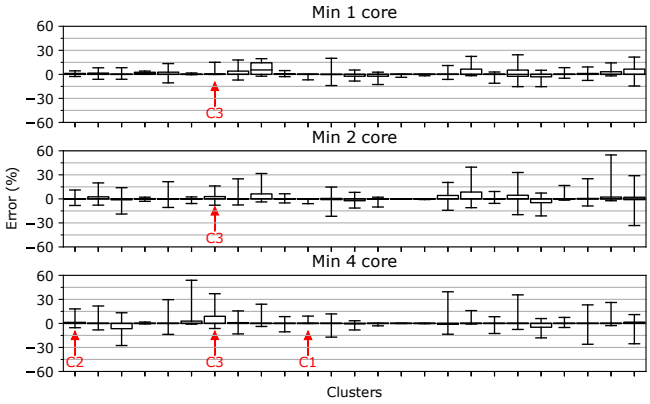


Figure 15: Prediction errors for the 1-day survival rate, as a function of minimum size and cluster.

that the larger Harvest VMs get slightly more cores overall.

Results for all clusters. The results above illustrate the accuracy of our predictions for individual clusters. We now turn to results for all clusters. Figure 15 plots the errors (in boxplot format) when predicting 1-day survival rate with 95% confidence, for each minimum size and cluster. Each box ranges between the first and third error quartiles, with the line representing the mean error, whereas the whiskers extend out to the 2.5th and 97.5th percentiles. The clusters marked C1, C2, and C3 are those from Figures 12, 13, and 14, respectively. The vast majority of mean errors are around 0% and the bulk of the errors are lower than 20% for most clusters.

Figure 16 plots the error distribution of the survival rate without confidence intervals, for each horizon and minimum size, aggregated across all clusters. As expected, short-term predictions (*i.e.*, current, 1-hour) have the lowest errors. The short-term results show that >90% of the predictions have no errors and the worst predictions have an error under 15%. Interestingly, long-term predictions (*i.e.*, 1-month) tend to be more accurate than medium-term ones (*i.e.*, 1-day, 1-week). The reason is that small load changes have a larger impact when predicting medium-term survival, whereas they often get smoothed out in the long term.

The figure also shows that errors are balanced and there

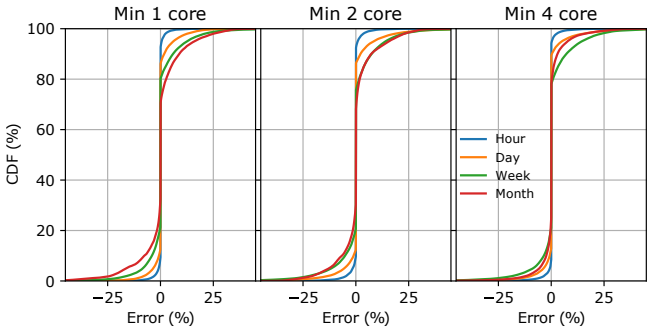


Figure 16: Prediction errors for time horizon and min size.

are not many more overpredictions than underpredictions (or vice-versa). When we average all the absolute errors, both current and 1-hour survival predictions exhibit errors lower than 2%. The 1-day and 1-week average absolute errors are roughly 6%, whereas the 1-month average absolute error is just under 4%.

Our predictions of the average number of cores available to Harvest VMs are even more accurate: the average error is smaller than 2.3% (<0.9% considering the confidence interval). When predicting the median and the 75^{th} percentile numbers of cores, the errors are below 4.1% (<1.5% considering the confidence interval). Even though our current Harvest VM implementation does not harvest memory, targeting our model to predict the memory available for harvesting also produces accurate results: the average absolute error is smaller than 1.5% (<0.5% considering the confidence interval).

Prediction adaptability. During this work, our model has been exposed to three versions of the VM scheduler that changed the allocation behaviors over time. We periodically re-train our model to capture these new behaviors. In addition, the auto-regressive features are especially good at adjusting to such changes. We have also evaluated our predictions with multiple hardware generations (our results are for the two most popular ones) and the model is able to adjust to them.

In summary, our predictions for both survival rate and average number of cores are *accurate and robust* for a wide range of cluster characteristics and behaviors.

Impact of the ML model. For comparison against our Random Forest model [24], we evaluated a Multi-Layer Perceptron (MLP) [35], Gradient Boosting [17], Exponential Smoothing (ETS) [18], and ARIMA [45]. MLP is a type of neural network. It achieves the closest results to our model, but we had to explore multiple combinations in the numbers of layers and neurons per layer. Figure 17 shows a comparison between the prediction errors of the two models' survival rates (left) and number of cores (right). The predictions are slightly worse using MLP and it does not provide confidence intervals. Training times are not a concern for large cloud providers. For context, one Random Forest training session typically takes around 16 hours on one Harvest VM, using 45 days of data from all 25 clusters. A similar MLP training session takes much longer, but we did not try using GPUs for



Figure 17: Avg absolute errors for Random Forest and MLP.

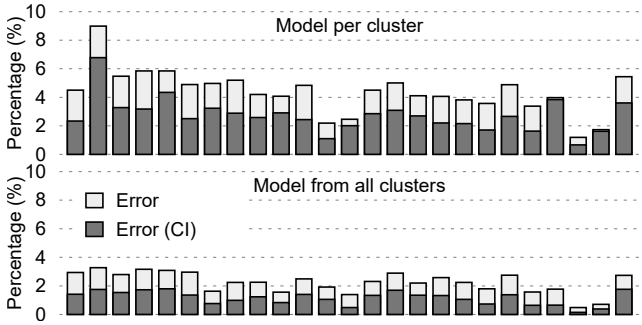


Figure 18: Avg absolute errors using two model types.

it. Prediction times are negligible in both cases.

The other models do not compare as well. Random Forests outperformed Gradient Boosting in both accuracy and performance. ETS and ARIMA are well-behaved for certain clusters and provide confidence intervals. However, they cannot incorporate other features (*e.g.*, cluster characteristics) that improve predictions in unseen situations. In contrast, Random Forests can easily use data from other clusters when the cluster starts to behave differently from its past (*e.g.*, the load increase in Figure 13). Nevertheless, our approach does incorporate the auto-regressive and moving average aspects of ARIMA.

Impact of the features. Our model uses 31 features. Both SHAP analysis [28] and a feature importance algorithm indicate that the auto-regressive features are the most relevant. However, the other features do improve prediction quality, especially when the cluster starts to behave differently. The features that we discarded do not improve our predictions.

Impact of global modeling. We can see the benefit of using data from other clusters by comparing prediction errors from 25 per-cluster models vs a single model trained with data from all 25 clusters. Figure 18 shows this comparison for survival rates. The top graph shows the errors of the independent models and the bottom one the errors for the single model. The bottom graph shows lower errors in every case.

7.4 Harvest VMs and Harvest Hadoop

As we mention in Section 7.1, we experiment with Harvest VMs and Harvest Hadoop in the Azure's production infrastructure, using two clusters called private and canary.

Private cluster. We have been running Harvest VMs and Harvest Hadoop in this cluster in production for more than 6 months. The organic regular-VM workload is a key-value

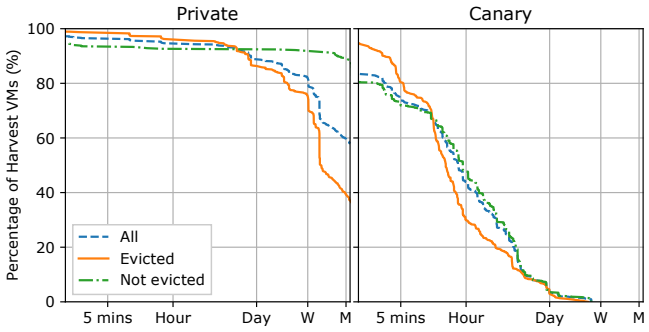


Figure 19: Time to evict Harvest VMs in the two clusters.

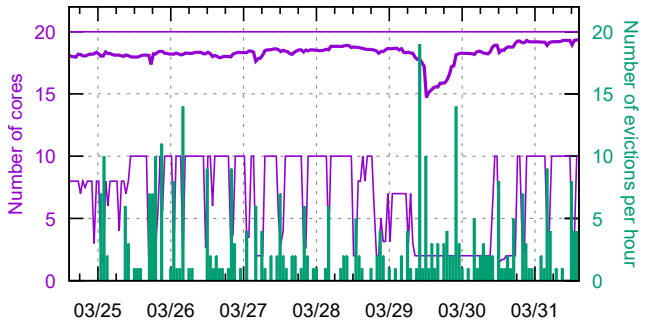


Figure 20: Minimum, maximum, and average number of harvested cores and evictions over one week.

store that uses full-server VMs but leaves many other servers empty. Our Harvest VMs (mininum 2 cores) run on these servers and execute Hadoop-based ML and data analytics jobs. As the Harvest VMs run alone, they grow to consume all the cores on their host servers. The left side of Figure 19 shows their survival statistics over 3 months. The "Not Evicted" VMs are those that terminated, instead of being evicted. Roughly 95% of the Harvest VMs that are evicted survive one hour or more, and roughly 40% survive for one month. These durations match the production jobs nicely: this cluster ran 105k tasks in a month, 90% of them ran for less than 10 minutes, 95% less than 1 hour, and only 1% longer than 6 hours. Interestingly, roughly 90% of the not evicted Harvest VMs last for more than a month. Over time, the organic load has increased and we now have capacity for around 450 Harvest VMs.

Canary cluster. To stress Harvest Hadoop, we create full deployments in the canary cluster, whose organic workload also seeks to stress the platform and varies significantly. Each deployment includes 100 workers in Harvest VMs of minimum size 2, and executes various Hadoop benchmarks. The results over 3 months appear on the right side of Figure 19. The constantly varying organic load and our stress-benchmarking result in only roughly 30% of the Harvest VMs that are evicted surviving one hour or longer. Even the ones that are not evicted are very short, and terminate before they are evicted.

To see these behaviors in more detail, we let a 100-VM setup run for over a week, continuously executing the TeraGen and Terasort benchmarks with 1000 map tasks. From these experiments, Figure 20 shows the minimum, maximum, and average number of cores the Harvest VMs got over the week. Most of the time, the Harvest VMs got over 18 cores on average. On March 29^{th} there was a surge in load and the average dropped to 15 cores. During this time, there was at least one Harvest VM with only 2 cores. The figure also shows the number of evictions per hour. During the load surge, there were 19 evictions but in 30% of the hours there were no evictions and in 75% there were 2 or fewer. After every eviction, the auto-scaler replaced the evicted Harvest VM with a new one trying to keep 100 workers at all times. For most of the VM re-creations, accesses to the remote storage service were not needed to re-hydrate the cache, because the data could come from other cached replicas; the exceptions were the 19 evictions during the load surge, which actually lost all cached replicas of certain files.

7.5 Cost comparisons

To compute the cost savings that Harvest VMs can accrue, compared to standard evictable and regular VMs, we can use the formula from Section 4. To use it, we need to instantiate the time to recover from evictions in core×hours (*recovery core hrs*), the prices per core×hour (*price* and α), and the number of minimum (*minsize core hrs*) and additional (*additional core hrs*) Harvest VM core×hours. The number of cores used by regular and standard evictable VMs as equivalent to the minimum size of Harvest VMs.

We compute the recovery time for each evicted VM from the experiments with the canary cluster. Specifically, an eviction forces the re-creation of the VM at another server, which takes roughly 30 seconds. In addition, Harvest Hadoop needs to re-create its workers, which takes a minimum of 2 minutes and 5 minutes at the 90^{th} percentile. The breakdown for the common case is 1 minute to get all the binaries (*e.g.*, Java, Hadoop, Docker, libraries), 30 seconds to setup and install dependencies, around 10 seconds to setup the environment (including security packages, compliance, and firewall setup), and around 10 seconds to start the services (DataNode and NodeManager) and heartbeat. Some stages are prone to long tails, which usually occur when creating a few hundred Harvest VMs at the same time. For this analysis, we assume that each eviction causes 5.5 minutes of recovery time.

To instantiate the prices, we use the amounts that Azure charges for the VMs from which we derive the Harvest VM resource quantities. Specifically, we instantiate the prices as \$0.126/(core×hour) for a regular VM and \$0.019/(core×hour) for a standard evictable VM [5, 6]. We use the latter price for the minimum size of the Harvest VMs as well. By default, we assume that the discount for additional Harvest VM cores (beyond the minimum size) over the evictable core price is 50%, *i.e.* $\alpha = 0.5$. Below, we discuss other values as well. As Harvest Hadoop can use as many cores as Harvest VMs give it, it is immaterial whether the provider charges for additional cores per-use or per-allocation.

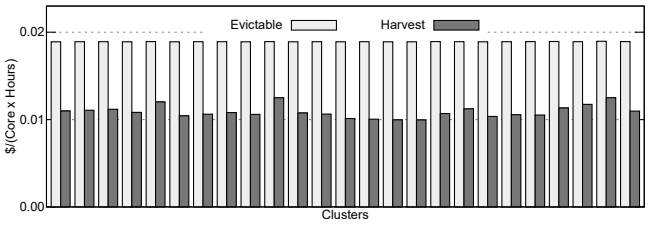


Figure 21: Costs when consuming the unallocated resources of 25 production clusters.

Using data from the simulations in Section 7.2 (where we consume as much of the unallocated resources as possible), we combine the recovery time above with the average survival rates for the 25 clusters and compute an average per-VM recovery overhead of only 0.13%. Again for the 25 clusters, we find that Harvest VMs receive an average of 7.2 additional cores beyond the 1-core minimum.

With these parameters instantiated, we compute the cost per useful core of Harvest VMs in the 25 clusters to range from 34% to 47% cheaper than standard evictable VMs. On average, Harvest VMs are 42% cheaper at \$0.011/(core×hour). Figure 21 illustrates these costs. The fact that evictable VMs suffer many more evictions is a minor factor in these savings, since survival times are much larger than recovery times for both VM classes. Instead, the key reason for the large savings is the additional cores that can be harvested at discounted prices. When those cores are priced the same as the minimum size ($\alpha = 1$), there are almost no savings. When they are free ($\alpha = 0$), Harvest VMs cost \$0.003/(core×hour) on average across the clusters, *i.e.* a savings of 84%.

Compared to filling the unallocated capacity of the 25 clusters with 1-core regular VMs, Harvest VMs are 91% cheaper on average for $\alpha = 0.5$. Here, the heavily discounted nature of evictable cores dominates. Lower prices for the additional cores increase these savings, whereas charging the same price as for evictable/minimum cores lowers the savings to 85%.

8 Lessons from production deployment

Adapting applications and fast adoption. We initially thought that the main users of Harvest VMs would be those who could deploy lots of evictable VMs. However, after discussing with internal teams, we soon realized that their workloads could not benefit from additional cores without modification. This made adoption harder, despite the much lower price of Harvest VMs. Fortunately, many large users at the provider rely on the Hadoop stack, so we devised Harvest Hadoop. These users then immediately and transparently adopted Harvest VMs. We are now starting to adapt a FaaS platform and Kubernetes for Harvest VMs.

Harvesting without evictions. Other potential users were concerned about experiencing frequent evictions. They were the motivation for our SLOs. Still, some would prefer not to have any evictions. For them, we are considering regular-priority Harvest VMs, which still have a (non-evictable) minimum size but can grow. For these VMs, the discount will apply only to the cores used beyond the minimum size.

Unbalanced Harvest VMs. Our current implementation only harvests cores, which may lead to VMs that cannot use some cores because their memory becomes too small. For example, some production VMs harvest 20 cores with only a fixed 16GB of memory. This imbalance is fine for some workloads but not others. Based on this, we started prototyping harvesting of unallocated memory. Another option is to define Harvest VM types with larger (fixed) memories, but that would make it harder to place them. The other resources have not posed imbalance problems so far.

Multiple Harvest VMs per server. Our implementation allows one Harvest VM per server because this works well for our initial (Hadoop) customers. However, to address the imbalance above and enable workloads that have less parallelism per VM, we are implementing the ability for users to specify a maximum size for each Harvest VM, and the fair sharing of a server's unallocated cores across multiple Harvest VMs. Our models easily extrapolate to having multiple of them per server. We need to add the maximum size of each Harvest VM and the number of Harvest VMs in the cluster as features.

New VM family. We initially limited Harvest VMs to a few pre-defined sizes (Section 4). However, some users needed VMs with faster disks or a different hardware generation. So, we had to create new types. For this reason, we plan to make harvesting a feature that can be enabled for most VM types, instead of being a separate family.

Impact on regular VM creation times. Our initial implementation of core reassignments had the unexpected side-effect that regular VM creation could be slowed down significantly on servers that were already hosting many VMs. The problem only became noticeable when we started testing in the canary cluster. Fixing it involved using a different API to the hypervisor and made the overhead negligible.

9 Conclusion

In this paper, we first characterized the unallocated resources of a large cloud provider. We then proposed to dynamically harvest the unallocated resources using Harvest VMs. To provide SLOs for these resources, we built an accurate ML-based predictor for VM survival rates and average number of cores. To demonstrate the use of Harvest VMs, we built a cluster scheduling framework called Harvest Hadoop. Finally, we discussed the lessons and results from our production deployment of Harvest VMs and Harvest Hadoop in Azure.

Acknowledgements

We thank Rebecca Isaacs, our shepherd, the anonymous reviewers, David Irwin, and Stanko Novakovic for their many helpful comments and suggestions.

References

[1] Amazon Elastic Compute Cloud. Amazon EC2 Spot Instances, 2019. https://aws.amazon.com/ec2/spot/.

[2] Pradeep Ambati and David Irwin. Optimizing the Cost of Executing Mixed Interactive and Batch Workloads on Transient VMs. In *SIGMETRICS*, 2019.

[3] Apache. Apache Hadoop, 2020. https://hadoop.apache.org/.

[4] Microsoft Azure. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[5] Microsoft Azure. Ev3 and Esv3-series. https://docs.microsoft.com/en-us/azure/virtual-machines/ev3-esv3-series.

[6] Microsoft Azure. Pricing Calculator. https://azure.microsoft.com/en-us/pricing/calculator/.

[7] Microsoft Azure. Introducing B-Series, Our New Burstable VM Size, 2019. https://azure.microsoft.com/en-us/blog/introducing-b-series-our-new-burstable-vm-size/.

[8] Microsoft Azure. Azure Spot Virtual Machines, 2020. https://azure.microsoft.com/en-us/pricing/spot.

[9] O. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Transactions on Economics and Computation (TEAC)*, 1(3), 2013.

[10] Leo Breiman. Random Forests. *Machine learning*, 45(1):5–32, 2001.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[12] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-Term SLOs for Reclaimed Cloud Computing Resources. In *SoCC*, 2014.

[13] Amazon Elastic Compute Cloud. Burstable Performance Instances, 2019. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[14] Google Cloud. Preemptible VM Instances, 2020. https://cloud.google.com/compute/docs/instances/preemptible.

[15] Intel Corp. Intel® CAT. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html.

[16] Eli Cortez, Anand Bonde, Alexandre Muzi, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*, 2017.

[17] Jerome H Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, pages 1189–1232, 2001.

[18] Everette S Gardner Jr. Exponential Smoothing: The State of the Art—Part II. *International Journal of Forecasting*, 22(4):637–666, 2006.

[19] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM Allocation Service at Scale. In *OSDI*, 2020.

[20] Apache Hadoop HDFS. HDFS DataNode Admin Guide, 2020. https://hadoop.apache.org/docs/current3/hadoop-project-dist/hadoop-hdfs/HdfsDataNodeAdminGuide.html.

[21] Apache Hadoop HDFS. HDFS Provided Storage, 2020. https://hadoop.apache.org/docs/current3/hadoop-project-dist/hadoop-hdfs/HdfsProvidedStorage.html.

[22] Kubernetes. Production-Grade Container Orchestration, 2020. https://kubernetes.io/.

[23] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *EuroSys*, 2014.

[24] Andy Liaw, Matthew Wiener, et al. Classification and Regression by Random Forest. *R News*, 2(3):18–22, 2002.

[25] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-Chun Feng, Mark Gardner, and Zhe Zhang. MOON: MapReduce On Opportunistic eNvironments. In *HPDC*, 2010.

[26] Michael J Litzkow, Miron Livny, and Matt W Mutka. Condor-A Hunter of Idle Workstations. In *ICDCS*, 1988.

[27] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

[28] Scott M Lundberg and Su-In Lee. A Unified Approach to Interpreting Model Predictions. In *NIPS*, 2017.

[29] Microsoft. Hyper-V Technology Overview, 2016. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview.

[30] Microsoft. Hyper-V Integration Services, 2019. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/integration-services.

[31] X. Ouyang, D. Irwin, and P. Shenoy. SpotLight: An Information Service for the Cloud. In *ICDCS*, 2016.

[32] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/.

[33] Prateek Sharma, Ahmed Ali-Edlin, and Prashant Shenoy. Resource Deflation: A New Approach For Transient Resource Reclamation. In *EuroSys*, 2019.

[34] Supreeth Shastri, Amr Rizk, and David Irwin. Transient Guarantees: Maximizing the Value of Idle Cloud Capacity. In *SuperComputing*, 2016.

[35] Bruce W Suter. The Multilayer Perceptron as an Approximation to a Bayes Optimal Discriminant Function. *IEEE Transactions on Neural Networks*, 1(4):291, 1990.

[36] Sean J Taylor and Benjamin Letham. Forecasting at Scale. *The American Statistician*, 72(1):37–45, 2018.

[37] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[38] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *EuroSys*, 2015.

[39] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *ISCA*, 2013.

[40] Apache Hadoop YARN. Dynamic Resource Configuration. https://issues.apache.org/jira/browse/YARN-999.

[41] Apache Hadoop YARN. In case of long running tasks, reduce node resource should balloon out resource quickly by calling preemption API and suspending running task. https://issues.apache.org/jira/browse/YARN-999.

[42] Apache Hadoop YARN. Graceful Decommission of YARN Nodes, 2020. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/GracefulDecommission.html.

[43] Apache Hadoop YARN. Opportunistic Containers, 2020. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/OpportunisticContainers.html.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[45] G Peter Zhang. Time Series Forecasting Using a Hybrid ARIMA and Neural Network Model. *Neurocomputing*, 50:159–175, 2003.

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

[47] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Í Goiri, and Ricardo Bianchini. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *OSDI*, 2016.

The *CacheLib* Caching Engine: Design and Experiences at Scale

Benjamin Berg[1], Daniel S. Berger[1,3], Sara McAllister[1], Isaac Grosof[1], Sathya Gunasekar[2], Jimmy Lu[2], Michael Uhlar[2], Jim Carrig[2], Nathan Beckmann[1], Mor Harchol-Balter[1], and Gregory R. Ganger[1]

[1]Carnegie Mellon University, [2]Facebook, [3]Microsoft Research

Abstract

Web services rely on caching at nearly every layer of the system architecture. Commonly, each cache is implemented and maintained independently by a distinct team and is highly specialized to its function. For example, an application-data cache would be independent from a CDN cache. However, this approach ignores the difficult challenges that different caching systems have in common, greatly increasing the overall effort required to deploy, maintain, and scale each cache.

This paper presents a different approach to cache development, successfully employed at Facebook, which extracts a core set of common requirements and functionality from otherwise disjoint caching systems. *CacheLib* is a general-purpose caching engine, designed based on experiences with a range of caching use cases at Facebook, that facilitates the easy development and maintenance of caches. CacheLib was first deployed at Facebook in 2017 and today powers over 70 services including CDN, storage, and application-data caches.

This paper describes our experiences during the transition from independent, specialized caches to the widespread adoption of CacheLib. We explain how the characteristics of production workloads and use cases at Facebook drove important design decisions. We describe how caches at Facebook have evolved over time, including the significant benefits seen from deploying CacheLib. We also discuss the implications our experiences have for future caching design and research.

1. Introduction

Large web services rely on caching systems to achieve high performance and efficiency. For example, at Facebook, CDN caches serve 70% of web requests, reducing latency by an order of magnitude. A single caching server can replace tens of backend database servers by achieving 20× higher throughput and hit ratios exceeding 80%.

At Facebook, a wide variety of caching systems form an integral part of the system architecture. Facebook's architecture includes CDN caches, key-value application caches, social-graph caches, and media caches (Figure 1). Caching plays a similar role at Amazon [26], Twitter [42, 92], Reddit [33, 89], and many other large web services.

Caching systems at Facebook. Historically, each caching system at Facebook was implemented separately. For example, Facebook separately designed CDN caches [86], key-value caches [72], social-graph caches [17], storage caches [71], database caches [2], and many others. The belief was that each of these highly specialized systems required a highly specialized cache in order to implement complex consistency protocols, leverage custom data structures, and optimize for a desired hardware platform.

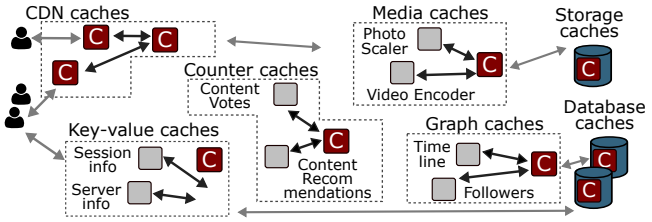


Figure 1: *Large web services rely on caching in many subsystems to improve system performance and efficiency.*

Although these caching systems serve different workloads and require different features, they share many important engineering and deployment challenges (Section 2). All of these systems process millions of queries per second, cache working sets large enough to require using both flash and DRAM for caching, and must tolerate frequent restarts due to application updates, which are common in the Facebook production environment. As the number of caching systems at Facebook increased, maintaining separate cache implementations for each system became untenable. By repeatedly solving the same hard engineering challenges, teams repeated each other's efforts and produced redundant code. Additionally, maintaining separate caching systems prevented the sharing of efficiency gains from performance optimizations between systems.

Hence, Facebook was faced with a tradeoff between generality and specialization. A more general-purpose caching solution might lose some domain-specific optimizations for individual systems, but it could reduce development overhead and increase synergistic efficiency between systems. The desire to balance this tradeoff gave rise to CacheLib, the general-purpose caching engine.

This paper describes Facebook's solution for scalable cache deployment: *CacheLib*. CacheLib is a C++ library that provides a common core of cache functionality, including efficient implementations of cache indexes, eviction policies, and stability optimizations for both DRAM and flash caches. CacheLib exposes its features via a simple, thread-safe API that allows programmers to easily build and customize scal-

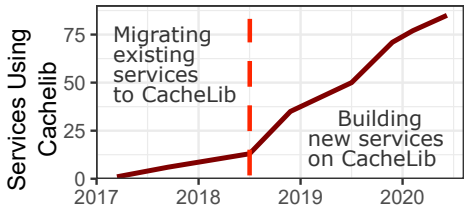


Figure 2: *The number of Facebook services built using CacheLib over time. For example, one service is Facebook's key-value caching system. Initial growth was due to migration of existing systems, but more recently, many new systems are built using CacheLib.*

able, highly concurrent caches. CacheLib is used both to build standalone caching systems and to add in-process caches to applications. Facebook's CDN, social-graph cache, application look-aside cache, and block-storage system all use caches built and customized using CacheLib.

CacheLib has been deployed in production since 2017 and today powers over 70 different services. Figure 2 shows CacheLib's growth during this period. Initially, CacheLib replaced existing caches, but since mid-2018 it has facilitated an explosive growth in caches throughout Facebook, leading to a significant reduction in backend load.

1.1. Lessons Learned from CacheLib

Developing an effective general-purpose caching framework has required not only understanding common caching use cases within Facebook, but also understanding more general trends in how caches will be deployed in the future. This section describes instances where the conventional wisdom about caching does not match our experience with Facebook's production environment.

Specialized caching systems can and should be built using a general-purpose caching engine. At Facebook, CacheLib has replaced existing specialized caches in several major services and has spurred the adoption of caches in many new applications. CacheLib offers a broader feature set than any single specialized caching system. Having a common core of features saves tens of thousands of lines of redundant code, improves system stability, and makes it easy for developers to deploy and tune new caches. Moreover, CacheLib serves as an aggregation point for optimizations and best practices. As a result, systems built on CacheLib achieve peak throughputs of a million requests per second on a single production server and hit ratios between 60 and 90%. To achieve this performance, each caching system customizes its cache to use their desired subset of CacheLib's features. To accommodate as many systems as possible, CacheLib's feature set has grown over time. As a result, features that once justified the construction of a specialized caching system are now available to any CacheLib-based system.

Production workloads require caching at massive scale. Prior workload studies of production systems [4, 5] have not shared the popularity distribution of keys. Consequently, popular benchmarking tools [24, 59] and academic papers [19, 36, 41, 49, 53, 65, 66, 68, 74, 90, 94] assume a Zipf popularity model with shape parameter $\alpha \approx .9$. This leads to the conclusion that DRAM-based caches are sufficient in most situations. We provide strong evidence that prior models have been too optimistic about the cacheability of production workloads.

Because workloads at Facebook are less cacheable than is generally assumed, caches at Facebook require massive capacities to achieve acceptable hit ratios. Caching systems at Facebook often comprise large distributed systems where each node has hundreds of gigabytes of cache capacity. This makes the use of flash for caching attractive. However, most caching systems have historically targeted DRAM and do not achieve acceptable performance using flash.

Caching is not a solved problem. CacheLib has continuously added features since its initial deployment in 2017 as new use cases and feature requests have come to light. These new features have seen rapid, widespread adoption from both existing CacheLib users and new systems developed using CacheLib, so that the common core of CacheLib features has grown with applications over time.

1.2. Bridging the Gap between Research and Production Caching Systems

Just as developers at Facebook had historically developed specialized caching systems, we note that the caching literature has often targeted specialized caching systems. This presents an obstacle to the uptake of ideas from the research community by industry, since the assumptions made by a specialized research system rarely align perfectly with the realities of a production environment. Our hope is that CacheLib can reduce these obstacles by providing a platform for the exploration of new ideas developed outside of Facebook. CacheLib and a selection of Facebook workloads will be open-sourced [1].

2. Motivation: Caching Use Cases

Large web services rely on hundreds of specialized services, which contain diverse use cases for caching. This section describes the caching needs of a sample of six production systems at Facebook.

Hierarchical and geo-distributed caches. Facebook's CDN focuses on serving HTTP requests for static media objects such as photos, audio, and video chunks from servers in user proximity. Specifically, a goal of CDN servers deployed outside of Facebook's network is to reduce the number of bytes sent over the wide-area network (byte miss rate). There are also CDN servers within Facebook's data centers; their goal is to reduce the number of backend and storage queries (object miss rate). Each CDN server uses a local cache, spanning both DRAM and flash.

Application look-aside caches. Web applications have a wide range of caching needs. For example, applications must cache database queries, user data, and usage statistics. Provid-

[1] For more information, visit `www.cachelib.org`

ing a specialized caching service for each application would be inefficient and hard to maintain. Thus, applications use RPCs to access a set of shared caching services. Each caching service consists of a large distributed system of caches.

In-process caches. Many applications cannot tolerate the RPC overhead of a remote cache. CacheLib makes it easy for these applications to include high-performance, in-process caches that are decoupled from the application logic.

For example, some backend applications use a CacheLib cache to store client session information which is used to rate-limit clients. Specifically, these applications cache flow counters that see very high bursts in request rates but can be evicted once a flow slows down. In this case, the latency and bandwidth requirements of the cache make remote caches infeasible. Instead, applications instantiate a CacheLib cache which provides zero-copy access to cached flow counters.

Machine-learning-model serving systems. User-facing machine-learning applications benefit from caching in multiple places. First, models often use inputs based on how users interact with content (e.g., liking a piece of content). Content interaction counters are thus cached so applications can quickly access the inputs required to generate a prediction (e.g., ranking content). Second, because repeatedly generating predictions based on the same inputs is computationally expensive, model predictions are also cached.

Storage-backend cache. Facebook uses large clusters of servers with spinning disks to store blocks of persistent data. Even with several caching layers in front of the block storage servers, some blocks remain popular enough to exceed the target IOPS of the disks. Storage servers use flash drives to cache popular blocks and shield the spinning disks from load. To support byte-range requests and append operations, these flash caches are tightly integrated in the storage-system stack.

Database page buffer. Data structures and small objects are stored in a variety of database systems. Database systems use page caches to increase their throughput and decrease access latencies. To support consistency and transactional operations, page caches are tightly integrated with database logic.

Across Facebook, we find hundreds of different services which implement a cache or whose efficiency could benefit from a caching layer. These use cases span all layers of the data-center stack and administrative domains. Research on caching spans operating systems [16,52], storage systems [20, 58], distributed systems [8,22,66], network systems [9,65], databases [30], and computer architecture [7,56,91].

CacheLib handles these diverse use cases by providing a *library* of components that makes it easy to rapidly build performant caches. In many cases, CacheLib caches have replaced highly specialized caching systems at Facebook. CacheLib is currently used in dozens of production systems, spanning five of the six examples described above. Notably, CacheLib is not currently used as a database page buffer (see Section 6). Hence, while CacheLib will not replace every specialized caching system, we have seen significant adoption of a general-purpose caching engine at Facebook.

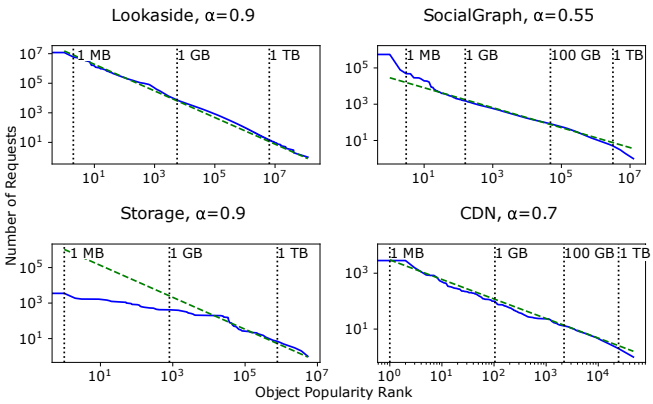


Figure 3: ***Many services are hard to cache.*** *Each graph plots the number of requests per object as a function of object popularity rank (log-log) for four production caching systems at Facebook. The green dashed line shows the best-fit Zipf distribution for each workload. Lower values of* α *indicate that a workload is harder to cache, hence SocialGraph and CDN are harder to cache. Storage is not Zipfian. Each sample consists of requests taken over 24 hours. The black dashed vertical lines along the x-axis show the cumulative size of the popular objects to the left of the line.*

3. Shared Challenges Across Caching Systems at Facebook

Despite the diversity of use cases for caching, our experience scaling and maintaining caching systems has revealed a set of core challenges that frequently overlap between use cases. This section describes the common challenges at Facebook.

The data in this section was gathered between December 2019 and May 2020 from 4 important workloads from a variety of caching use cases (Section 2). The *Lookaside* and *SocialGraph* systems are both from application-data caches. *Lookaside* is a service which provides on-demand caching to a variety of applications. *SocialGraph* is specifically used to cache information from the Facebook social graph. The *Storage* system is a storage backend cache, and *CDN* is a cache in Facebook's CDN. Each workload represents the traffic to one machine within its respective service.

3.1. Massive Working Sets

One central challenge at Facebook is massive *working sets*. A working set describes the set of objects in a workload which are popular enough to benefit from caching. A workload with a larger working set requires a larger cache to produce the same hit ratio as a workload with a smaller working set.

To measure working sets, one must account for both the amount of popular data seen over time and the rate of change in the popularity of data over time. Therefore, we present both popularity distributions and churn rates at Facebook.

Popularity. The popularity distribution of a workload measures the frequency of each key over some time horizon

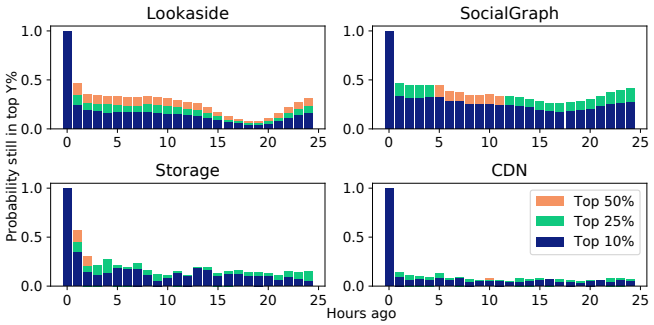


Figure 4: ***Object popularity changes rapidly over time.*** *Each graph plots the probability that the top 10%-most-requested objects remain popular after x hours. Across all workloads, there is a significant drop off in popularity after even a single hour.*

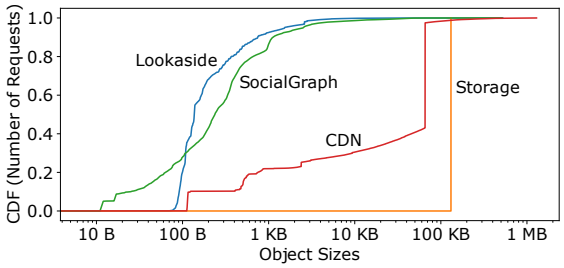


Figure 5: ***Object sizes vary widely and small objects are common.*** *Distribution of value sizes for all four workloads. Object size is shown on the X-axis on a log scale. The Y-axis shows a complimentary CDF – the fraction of requests for objects which are less than a given size. Object sizes are small in the Lookaside and SocialGraph workloads. Storage and CDN split objects greater than 64 KB and 128 KB, respectively, across multiple keys.*

in a sampled trace. These frequencies indicate the relative popularity of different objects in the system. Prior measurements of CDN and web workloads indicate that highly-skewed Zipf distributions are a common popularity distribution [5, 14, 24, 38, 41, 48, 83, 85]. Informally, in a Zipf distribution "the most popular 20% of objects account for 80% of requests". Formally, in a Zipf distribution the i-th most popular object has a relative frequency of $1/i^{\alpha}$. While some studies indicate α as low as 0.56 [38, 40], most prior measurements indicate $0.9 < \alpha \leq 1$ [5, 14, 24, 41, 48, 85]. This parameter range has become the standard evaluation assumption in many recent system papers [19, 36, 41, 49, 53, 65, 66, 68, 74, 90, 94].

Figure 3 shows the popularity distributions on log-log scale for four workloads at Facebook. At this scale, a Zipf distribution would be a straight line with negative slope ($-\alpha$). *Lookaside* is the only system of the four whose popularity distribution is Zipfian with α close to 1. *Storage*'s distribution is much flatter at the head of the distribution, even though the tail follows a Zipf distribution. Furthermore, although Zipfian, *SocialGraph* and *CDN*'s distributions exhibit $\alpha = 0.55$ and $\alpha = 0.7$, respectively. Lower α means that a significantly higher proportion of requests go to the tail of the popularity distribution, which leads to a larger working set.

Churn. *Churn* refers to the change in the working set due to the introduction of new keys and changes in popularity of existing keys over time. The popular YCSB [24] workload generator assumes that there is no churn, i.e., each key will remain equally popular throughout the benchmark. This benchmark and the no-churn assumption is used widely in the evaluation of system papers [19,36,49,53,65,66,68,74,90,94].

In Facebook production workloads, we find a high degree of churn. We define an object to be *popular* if it is among the 10% of objects that receive the most requests. Figure 4 shows how the set of popular objects changes over time. For example, the blue bar at $x = 3$ shows the probability that an object which was popular 3 hours ago is still in the top 10%-most-requested objects. Across all workloads, over two-thirds of popular objects in a given hour fall out of the top 10% after just one hour. Such high churn applies independent of which hour we use as the baseline, for different percentiles (e.g., top 25%), and with different time granularities (e.g., after 10 minutes, 50% of popular objects are no longer popular). This high churn rate increases the importance of temporal locality and makes it harder for caching policies to estimate object popularity based on past access patterns.

3.2. Size Variability

In addition to popularity and churn, object sizes play a key role in cache performance. Figure 5 shows the object size distribution for four large use case. For *Storage* and *CDN*, we find that 64KB and 128KB chunks, respectively, are very common, which result from dividing large objects into chunks. For *Lookaside* and *SocialGraph*, we find object sizes spanning more than seven orders of magnitude. Note the preponderance of small objects, which arise from graph edges/nodes, RPC computation results, and negative caching (see Section 4.3).

These findings restrict the design space for a general caching system. For example, many existing caching systems [3, 32, 35, 37, 70, 75, 79, 87] store at most a single object per cache line (64B). For a system such as *SocialGraph*, where a significant fraction of objects are between 10B and 20B, this approach wastes space. Another challenge is in-memory data structures which are used as an index for objects on flash. The per-object overhead differs across existing systems between 8B and 100B [32, 37, 70, 79, 86, 87]. For a system with a median object size of 100B, such as *Lookaside*, this means that 80GB - 1TB of DRAM is needed to index objects on a 1TB flash drive. It is imperative to handle highly variable object sizes while limiting memory and storage overhead.

3.3. Bursty Traffic

Another common theme is that Facebook's traffic is quite bursty. Figure 6 shows the actual request arrival rate compared to a Poisson arrival sequence, which is often assumed in system evaluations [53, 66, 73, 74, 84]. Figure 6 shows that the actual arrival rate varies much more than Poisson suggests.

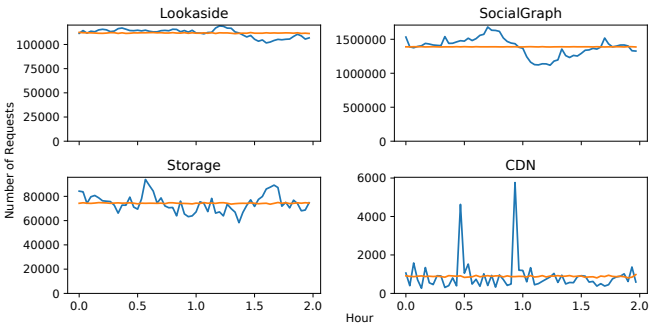


Figure 6: ***Requests are very bursty.*** *Number of requests (blue) for every two minutes over the 2 hour horizon compared to a Poisson arrival sequence (orange) with the same mean number of arrivals. The two hour window covers the peak traffic time within a day for each service. CDN has particularly high short-term bursts.*

This is particularly apparent for *CDN*, which has sharp bursts of traffic on top of a fairly steady request rate. Variable arrival rates make it hard to provision caching systems with sufficient resources to maintain low latencies during a load spike.

3.4. Resource Management

To be efficient, caches should make use of all available resources without exceeding them. This is particularly important for DRAM. A common deployment scenario includes CacheLib as well as application processes, and the kernel, all of which consume DRAM. As the workload composition or intensity changes, the memory available for caching can vary widely. For example, caches which handle variable-sized objects often have unpredictable levels of memory usage. Attempting to provision all available memory for caching can therefore lead to memory over-commitment, a well known challenge for in-process caches [78]. Specifically, a memory spike in the application might lead to the cache being dropped due to the kernel's out-of-memory (OOM) killer. Many open-source caching systems are not OOM-aware [87], leading to significant operational challenges [77]. CacheLib dynamically allocates and frees memory used by the cache to avoid these crashes without leaving too much unused memory.

3.5. Computationally Costly Query for Empty Results

Caching systems typically focus on tracking and storing results for valid backend queries. However, some use cases frequently send queries that have empty results, indicating that the requested object does not exist. This occurs often in database systems that track associations between users, where a user might query for the set of acquaintances they have in common with another user. Such queries are typically computationally costly for the backend database. For example, when querying the social graph, users frequently ask to see the set of associations they share with another user, and find that these associations do not exist. Hence, in *SocialGraph*'s workload, we measure that 55.6% of requests are for keys that do not exist. The remaining 44.4% of requests ask for valid objects, and the corresponding cache hit ratio among these requests is 86.5%. Failure to cache empty results would thus lower the cache hit ratio significantly.

3.6. Updating Cached Data Structures

Caches should efficiently support structured data. This is particularly important for in-process caches that directly interact with application data structures. For instance, the rate limiter described in Section 2 stores multiple fields in a single cache object. Applications often want the ability to update specific fields in a cached data structure without deserializing, updating, and re-serializing the object.

3.7. Frequent Restarts

Finally, production caches frequently restart in order to pick up code fixes and updates. This happens because engineering teams require the ability not only to roll out new code quickly, but to roll back changes quickly as well. For example, 75% of *Lookaside* caches and 95% of *CDN* caches have an uptime less than 7 days. Even systems such as *Storage* and *SocialGraph* which have longer uptimes on average follow a regular monthly maintenance schedule which requires restarting cache processes. Most caching systems are transient, meaning that their content is lost upon application restart [37,55]. Transience is problematic because large caches take a long time to warm up. It would take hours or even days for cache hit ratios to stabilize following the restart of a transient cache at Facebook. Prior work has suggested the alternative of warming a cache after restart, but this requires an explicit warm-up phase as part of routing [33, 72] or requires slow deployments [42].

3.8. Summary

While not every caching use case exhibits every challenge above, each use case does exhibit multiple of these challenges. We describe how CacheLib addresses these issues next in Section 4. The power of using a general-purpose caching engine to address these challenges is that all applications which use CacheLib have access to every CacheLib feature if and when it is needed.

4. Design and Implementation

CacheLib enables the construction of fast, stable caches for a broad set of use cases. To address common challenges across these use cases as described in Sections 2 and 3, we identify the following features as necessary requirements for a general-purpose caching engine.

Thread-safe cache primitives: To simplify programming for applications that handle highly bursty traffic, CacheLib provides a thread-safe API for reads, writes, and deletes. In addition, thread-safety simplifies the implementation of consistency and invalidation protocols. Concurrent calls to the CacheLib API leave the cache in a valid state, respect linearizablility [47] if referencing a common key, and incur minimal resource contention.

Transparent hybrid caching: To achieve high hit ratios while caching large working sets, CacheLib supports caches composed of both DRAM and flash, known as *hybrid caches*.

Hybrid caches enable large-scale deployment of caches with terabytes of cache capacity per node. CacheLib hides the intricacies of the flash caching system from application programmers by providing the same byte-addressable interface (Section 4.1) regardless of the underlying storage media. This transparency allows application programmers to ignore when and where objects get written across different storage media. It also increases the portability of caching applications, allowing applications to easily run on a new hardware configurations as they become available.

Low resource overhead: CacheLib achieves high throughput and low memory and CPU usage for a broad range of workloads (Section 2). This makes CacheLib suitable for in-process use cases where the cache must share resources with an application. Low resource overheads allow CacheLib to support use cases with many small objects.

Structured items: CacheLib provides a native implementation of arrays and hashmaps that can be cached and mutated efficiently without incurring any serialization overhead. Caching structured data makes it easy for programmers to efficiently integrate a cache with application logic.

Dynamic resource monitoring, allocation, and OOM protection: To prevent crashes from temporary spikes in system memory usage, CacheLib monitors total system memory usage. CacheLib dynamically allocates and frees memory used by the cache to control the overall system memory usage.

Warm restarts: To handle code updates seamlessly, CacheLib can perform *warm restarts* that retain the state of the cache. This overcomes the need to warm up caches every time they are restarted.

4.1. CacheLib API

The CacheLib API is designed to be simple enough to allow application programmers to quickly build in-process caching layers with little need for cache tuning and configuration. At the same time, CacheLib must scale to support complex application-level consistency protocols, as well as zero-copy access to data for high performance. Choosing an API which is both simple and powerful was an important concern in the design of CacheLib.

The API centers around the concept of an `Item`, an abstract representation of a cached object. The `Item` enables byte-addressable access to an underlying object, independent of whether the object is stored in DRAM or flash. Access to cached `Items` is controlled via an `ItemHandle` which enables reference counting for cached `Items`. When an `ItemHandle` object is constructed or destroyed, a reference counter for the corresponding `Item` is incremented or decremented, respectively. An `Item` cannot be evicted from the cache unless its reference count is 0. If an `Item` with a non-zero reference count expires or is deleted, existing `ItemHandles` will remain valid, but no new `ItemHandles` will be issued for the `Item`.

Figure 7 shows the basic CacheLib API. To insert a new object into the cache, `allocate` may first evict another `Item` (according to an eviction policy) as long as there are no outstanding `ItemHandles` that reference it. The new `Item` can be configured with an expiration time (TTL). It is created within the given memory "pool" (see below), which can be individually configured to provide strong isolation guarantees. Any new `Items` only become visible after an `insertOrReplace` operation completes on a corresponding `ItemHandle`.

```
ItemHandle allocate(PoolId id, Key key,
 uint32_t size, uint32_t ttlSecs = 0)
bool insertOrReplace(const ItemHandle& handle)
ItemHandle find(Key key)
void* Item::getMemory()
void* Item::markNvmUnclean()
bool remove(Key key)
```

Figure 7: *The CacheLib API uses an `Item` to represent a cached object, independent of whether it is cached in DRAM or on flash.*

To access cached `Items`, `find` creates an `ItemHandle` from a key, after which `getMemory` allows unsynchronized, zero-copy access to the memory associated with an `Item`. To atomically update an `Item`, one would `allocate` a new `ItemHandle` for the key they wish to update, perform the update using `getMemory`, and then make the update visible calling `insertOrReplace` with the new `ItemHandle`. Because CacheLib clients access raw memory for performance, CacheLib trusts users to faithfully indicate any mutations using the method `markNvmUnclean`. Finally, `remove` deletes the `Item` identified by a key, indicating invalidation or deletion of the underlying object.

```
struct MyType {int foo; char bar[10];}
TypedHandleImpl<Item, MyType> typedHandle{
 cache->find(..)};
```

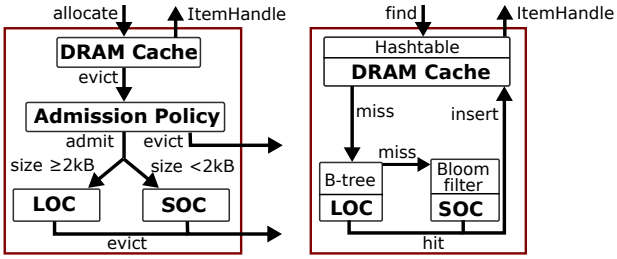
Figure 8: *Typed `ItemHandles` allow CacheLib to natively store structured objects. In addition to statically sized `Items`, CacheLib also supports variably sized `Items`. For example, CacheLib implements a hashmap that can dynamically grow, offer zero-copy access to its entries, and is treated as an evictable cache `Item`.*

Figure 8 shows a simple example of CacheLib's native support for structured data. Structured `Items` are accessed through a `TypedHandle`, which offers the same methods as an `ItemHandle`. `TypedHandles` enable low-overhead access to user-defined data structures which can be cached and evicted just like normal `Items`. In addition to statically sized data structures, CacheLib also supports variably-sized data structures; for example, CacheLib implements a simple hashmap that supports range queries, arrays, and iterable buffers.

CacheLib implements these APIs in C++, with binding to other languages such as Rust.

4.2. Architecture Overview

CacheLib is designed to be scalable enough to accommodate massive working sets (Section 3.1) with highly variable sizes (Section 3.2). To achieve low per-object overhead, a single



(a) `allocate` **(b)** `find`

Figure 9: *The* `find` *and* `allocate` *paths for a hybrid cache constructed using CacheLib.*

Item Metadata	DRAM Overhead	Data Type
Eviction policy state	12B	1x4B timestamp, 2x4B pointers
Item creation timestamp	4B	4B timestamp
Expiration time (for TTLs)	4B	4B timestamp
Key size + object size	4B	4B `size_t`
Reference counting	2B	13b public ref count, 3b internal ref count
Hash table chaining	4B	4B pointer
Flags	1B	8 binary flags

Table 1: *CacheLib's DRAM cache uses 31B of metadata per* `Item`*.*

CacheLib cache is composed of several subsystems, each of which is tailored to a particular storage medium and object size. Specifically, CacheLib consists of a DRAM cache and a flash cache. The flash cache is composed of two caches: the Large Object Cache (LOC) for `Items` ≥2KB in size and Small Object Cache (SOC) for `Items` <2KB in size.

An `allocate` request is fulfilled by allocating space in DRAM, evicting `Items` from DRAM if necessary. Evicted `Items` are either admitted to a flash cache (potentially causing another eviction) or discarded. A `find` request successively checks for an object in DRAM, then LOC, then SOC. This lookup order minimizes the average memory access time [46] of the cache (see Appendix A). A `find` call returns an `ItemHandle` immediately. If the requested object is located on DRAM, this `ItemHandle` is ready to use. If the requested object is located on flash, it is fetched asynchronously and the `ItemHandle` becomes ready to use once the object is in DRAM. An empty `ItemHandle` is returned to signify a cache miss. These data paths are summarized in Figure 9.

We now describe CacheLib's subsystems in more detail.

DRAM cache. CacheLib's DRAM cache uses a chained hash table to perform lookups. The DRAM cache can be partitioned into separate *pools*, each with its own eviction policy. Programmers select a particular `PoolId` when calling `allocate` (see Figure 7), allowing the isolation of different types of traffic within a single cache instance.

For performance, cache memory is allocated using slab classes [6, 22, 37] which store objects of similar sizes. CacheLib uses 4MB slabs and implements a custom slab allocator. Each slab requires 7B of DRAM (3B of internal metadata + 4B to identify the size of objects in the slab). Because CacheLib workloads often include many objects of a specific size (e.g., 80B), the sizes corresponding to each slab class are configured on a per-workload basis to minimize fragmentation. Further optimizations for objects smaller than 64B or larger than 4MB are described in Section 4.3.

Each slab class maintains its own eviction policy state. CacheLib is designed to support the continual development of new eviction policies, and currently supports LRU, LRU with multiple insertion points, 2Q [54, 93], and TinyLFU [31]. These eviction policies differ in their overheads and their biases towards either recency or frequency, and are thus configured on a per-workload basis as well. To approximate a global eviction policy, memory is rebalanced between slab classes using known rebalancing algorithms [72]. To support these policies, among other features, CacheLib dedicates 31B of DRAM overhead per item. Table 1 describes the metadata which comprises this DRAM overhead.

To guarantee atomic metadata operations, CacheLib relies on a variety of known optimization techniques [35, 62, 64], including fine-grained locking, user-space mutexes, and C++ atomics. This is particularly important for eviction policies, where naive implementations lead to lock contention and limit throughput [6, 9, 10, 61]. For example, under LRU, popular `Items` frequently compete to be reset to the most-recently-used (MRU) position. This is particularly common at Facebook due to our high request rates (see Figure 6). CacheLib adopts a simple solution to reduce contention: `Items` that were recently reset to the MRU position are not reset again for some time T [9, 87]. As long as T is much shorter than the time it takes an object to percolate through the LRU list (i.e., eviction age), this simplification does not affect hit ratios in practice. CacheLib also uses advanced locking mechanisms such as flat combining [45] to reduce resource contention.

Flash cache. When `Items` are evicted from the DRAM cache, they can optionally be written to a flash cache. Due to high popularity churn (Section 3), the content cached on flash changes continually. Hence, in addition to maintaining low per-object overhead, CacheLib must contend with the limited write endurance of flash cells.

To reduce the rate of writes to flash, CacheLib selectively writes objects to the flash cache. If an object exists on flash and was not changed while in DRAM, it is not written back to flash. Otherwise, CacheLib admits objects to flash according to a configurable admission policy. CacheLib's default admission policy is to admit objects to the flash cache with a fixed probability p [57]. Adjusting the probability p allows fine-grained control over write rate to flash. Section 5 describes our experience with more complex admission policies.

Another consideration for flash endurance is *write amplification* which happens when the number of bytes written to the device is larger than the number of bytes inserted into the cache. For instance, CacheLib performs extra writes to

store metadata and is forced to write at block granularities. We distinguish between *application-level* write amplification, which occurs when CacheLib itself writes more bytes to flash than the size of the inserted object, and *device-level* write amplification, which is caused by the flash device firmware. CacheLib's flash caches are carefully designed to balance both sources of write amplification and DRAM overhead.

The *Large Object Cache* (LOC) is used to store objects larger than 2KB on flash. Because LOC objects are larger than 2KB, the number of unique objects in a LOC will only number in the millions. It is therefore feasible to keep an in-memory index of the LOC. The LOC uses segmented B+ trees [23, 34, 63] in DRAM to store the flash locations of `Items`. `Items` are aligned to 4KB flash pages, so the flash location is a 4B, 4KB-aligned address. This allows the LOC to index up to 16TB of flash storage space.

The LOC uses flash to further limit the size of the DRAM index. Keys are hashed to 8B. The first 4B identify the B+-tree segment, and the second 4B are used as a key within in a tree segment to lookup a flash location. A hash collision in the DRAM index will cause CacheLib to believe it has found an object's flash location. Hence, LOC stores a copy of each object's full key on flash as part of the object metadata and validates the key after the object is fetched off flash. Each flash device is partitioned into regions which each store objects of a different size range. Hence, the object size can be inferred from where it is located on flash, without explicitly storing object sizes in DRAM. CacheLib can then retrieve the object via a single flash read for the correct number of bytes. To reduce the size of addresses stored in DRAM, every 4KB flash page stores at most a single object and its metadata. This is space-efficient because LOC only stores objects larger than 2KB. Objects larger than 4KB can span multiple pages. Because the LOC reads and writes at the page level, any fragmentation also causes application-level write amplification.

To amortize the computational cost of flash erasures, the LOC's caching policy evicts entire regions rather than individual objects. (Region size is configurable, e.g., 16MB.) By default, FIFO is used so that regions are erased in strictly sequential order [60]. Writing sequentially improves the performance of the flash translation layer and greatly reduces device-level write amplification (see Section 5.2). If FIFO eviction evicts a popular object, it may be readmitted to the cache [86]. Alternatively, LOC supports a pseudo-LRU policy which tracks recency at region granularity. A request for any object in a region logically resets the region to the MRU position. Evictions erase the entire region at the LRU position.

The *Small Object Cache* (SOC) is used to store objects smaller than 2KB on flash. Because billions of small objects can fit on a single 1TB flash device, an exact lookup index (with associated per-object metadata) would use an unreasonably large amount of DRAM [32]. Hence, SOC uses an approximate index that scales with the number of flash pages.

SOC hashes keys into sets.Each set identifies a 4KB flash page which is used to store multiple objects. Objects are evicted from sets in FIFO order. A naive implementation of this design would always read a set's page off flash to determine whether it contains a particular object. As an optimization, CacheLib maintains an 8B Bloom filter in DRAM for each set, each with 4 probes. This filter contains the keys stored on the set's flash page and prevents unnecessary reads more than 90% of the time [11, 12]. Figure 10 shows the `alloc` path, and Figure 11 shows the `find` path.

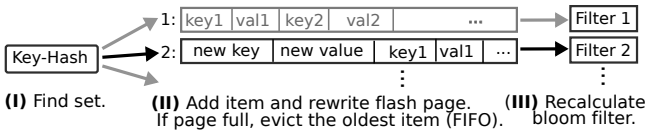


Figure 10: *SOC `alloc` proceeds by hashing into sets (I). CacheLib then rewrites the page (II), possibly evicting an object (following FIFO order). Finally, CacheLib recalculates the bloom filter with the `Items` currently stored in this set's 4KB page (III).*

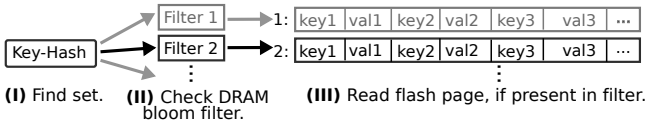


Figure 11: *SOC `find` proceeds by hashing into sets (I) and then checking a bloom filter, which indicates whether an object is likely to be stored on flash (II). If the bloom filter does not contain the key, the object is definitely not present on flash. Otherwise, CacheLib reads the 4KB flash cache and searches for the key (III).*

Controlling write amplification in the SOC is particularly challenging. Admitting an object to the SOC requires writing an entire 4KB flash page, and is thus a significant source of application-level write amplification. This also applies to the `remove` operation, which removes an object from flash. Similarly, because keys are hashed to sets, admitting a stream of objects to the SOC causes random writes that result in higher device-level write amplification. Furthermore, the SOC only supports eviction policies that do not require state updates on hits, such as FIFO, since updating a set on a hit would require a 4KB page write. These challenges highlight the importance of advanced admission policies (see Section 5.2).

4.3. Implementation of Advanced Features

CacheLib supports many applications with demanding requirements. To support these applications efficiently, CacheLib implements several advanced features, making them available to all CacheLib-based services under the same, general-purpose CacheLib API. We describe the implementation of four important features: structured items, caching large and small objects, warm restarts, and resource monitoring, corresponding to challenges already discussed in Section 3.

Structured items. Because CacheLib provides raw access to cached memory, flat data structures can be easily cached using the CacheLib API. In addition, CacheLib natively supports arrays and maps. CacheLib supports an `Array` type for fixed-size objects at no additional overhead for each entry in the array. The `Map` type supports variable object sizes, and comes

in ordered and unordered variants. The overhead for each `Map` entry is 4B to store its size.

Caching large and small objects in DRAM. To store objects larger than 4MB in size, CacheLib chains multiple DRAM `Items` together into one logical large item. This chaining requires an additional 4B `next` pointer per object in the chain. The most common use case for large objects is the storage of structured items. While it is uncommon for a single, logical object to be larger than 4MB, we frequently see `Arrays` or `Maps` that comprise more than 4MB in aggregate.

CacheLib also features *compact caches*, DRAM caches designed to cache objects smaller than a cache line (typically 64B or 128B). Compact caches store objects with the same key size and object size in a single cache line [18, 29, 46, 80]. Compact caches are set-associative caches, where each cache line is a set which is indexed by a key's hash. LRU eviction is done within each set by repositioning objects within a cache line. Compact caches have no per-object overhead.

One prominent example of using compact caches is CacheLib's support for *negative caching*. Negative cache objects indicate that a backend query has previously returned an empty result. Negative cache objects are small, fixed-size objects which only require storing a key to identify the empty query. As discussed in Section 3.5, negative caching improves hit ratios drastically in *SocialGraph*. Negative caching is not used by *Lookaside*, *Storage*, or *CDN*, but it is employed by 4 of the 10 largest CacheLib-based systems.

Both of these features reinforce CacheLib's overarching design, which is to provide specialized solutions for objects of different sizes in order to keep per-object overheads low.

Dynamic resource usage and monitoring. CacheLib monitors the total system memory usage and continuously adapts the DRAM cache size to stay below a specified bound. CacheLib exposes several parameters to control the memory usage mechanism. If the system free memory drops below `lowerLimitGB` bytes, CacheLib will iteratively free `percentPerIteration` percent of the difference between `upperLimitGB` and `lowerLimitGB` until system free memory rises above `upperLimitGB`. A maximum of `maxLimitPercent` of total cache size can be freed by this process, preventing the cache from becoming too small. Although freeing memory may cause evictions, this feature is designed to prevent outright crashes which are far worse for cache hit ratios (see Figure 15). As system free memory increases, CacheLib reclaims memory by an analogous process.

Warm restarts. CacheLib implements warm restarts by allocating DRAM cache space using POSIX shared memory [76]. This allows a cache to shut down while leaving its cache state in shared memory. A new cache can then take ownership of the cache state on start up. The DRAM cache keeps its index permanently in shared memory by default. All other DRAM cache state is serialized into shared memory during shutdown. The LOC B-tree index and SOC Bloom filters are serialized and written in a dedicated section on flash during shutdown.

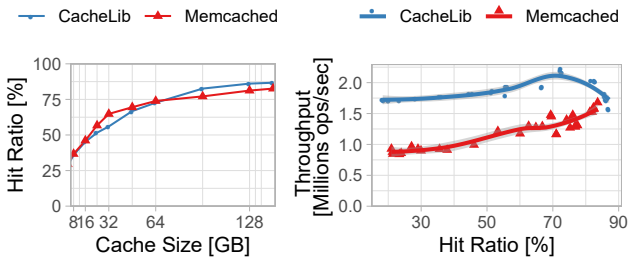


Figure 12: *A comparison of CacheLib to Memcached for a range of cache sizes. CacheLib and Memcached achieve similar hit ratios, but CacheLib achieves much higher throughput.*

5. Evaluation

In evaluating CacheLib, we aim to show that CacheLib's API and feature set is flexible enough to implement common use cases both inside and outside Facebook. Specifically, we show that CacheLib-based systems can easily achieve performance that is competitive with specialized solutions without requiring any specialization of the core caching engine. We also show how CacheLib's widespread adoption has had a significant impact on the Facebook production environment.

5.1. System Comparisons

We drive our experiments using *CacheBench*, the cache benchmarking tool that ships with CacheLib. For the sake of comparison, we extend CacheBench to target an in-process version of Memcached [37], as well as HTTP proxy (CDN) caches.

CacheBench provides a modular request generator by sampling from configurable popularity, key size, and object size distributions. To emulate churn, CacheBench continuously introduces new keys at a configurable rate. We instantiate these parameters from the measurements for the application look-aside and CDN use cases presented in Section 3.

Application look-aside cache. Before CacheLib was developed, several teams at Facebook used an internal variant of Memcached as a look-aside cache. However, applications now use a CacheLib-based look-aside cache. We therefore compare CacheLib to a minimally changed Memcached v1.6.6, which is the latest version and incorporates many recent optimizations. For fairness, we configure CacheLib and Memcached to both use their implementations of LRU eviction. To implement the look-aside pattern, CacheBench configures CacheLib to implement a "set" as an `allocate` followed by `insertOrReplace` and a "get" by `find` and a subsequent access to the `ItemHandle`'s `getMemory` method.

We evaluate CacheLib and Memcached on a range of cache sizes using 32 threads each. When the cache is small, the hit ratio is low, which stresses the eviction code paths (set operations). When the cache is large, the hit ratio is high, which stresses the LRU-head update code paths (get operations).

Figure 12 shows the hit ratios and throughputs for cache sizes between 8 and 144GB and a typical working set of 100 million objects. Memcached and CacheLib achieve similar hit

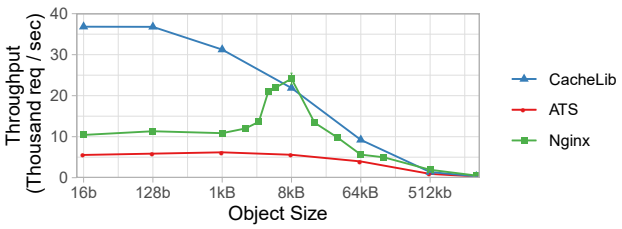


Figure 13: *Comparison of CacheLib to ATS and NGINX HTTP flash caching systems for different object sizes. CacheLib significantly improves throughput for most object sizes.*

ratios, with Memcached slightly higher at small cache sizes and slightly lower at large cache sizes. Across all cache sizes, CacheLib achieves higher throughputs than Memcached, processing up to 60% more requests per second than Memcached.

CacheLib's higher throughput is largely due to optimizations that reduce lock contention. For example, CacheLib uses flat combining (see Section 4.2) to reduce contention on the LRU list head. Also, CacheLib uses $T = 60$ seconds (see Section 4.2) in this experiment. For $T = 10$ seconds, CacheLib consistently outperforms Memcached's hit ratio, at the cost of lower throughput. In production, most deployments use the default $T = 60$ seconds.

HTTP server cache. Hybrid DRAM-flash caches are prevalent at Facebook. For example, hybrid caches are used as CDN proxy caches. We compare a CacheLib-based HTTP server cache to NGINX and Apache Traffic Server (ATS), which are widely used to build flash-based CDNs [1, 44, 69, 82]. The CacheLib implementation is a FastCGI server with an NGINX frontend. Each system uses its default configuration for a 512GB flash cache. The systems fetch misses from a high-performance origin backend that is never the bottleneck.

To illustrate the effect of object size on flash cache performance, we configured all object sizes to be equal and then repeated the experiment for a range of object sizes. To keep hit ratios constant across trials, we adjusted the number of unique objects to maintain a constant working set size.

Figure 13 shows that CacheLib's explicit handling of small objects for flash caching provides a sizable advantage over NGINX and ATS. As the object size becomes larger, this advantage wanes. Eventually object sizes become large enough that all three systems become network-bound and their throughputs drop precipitously.

We observe that NGINX performs particularly well when object sizes are between 4KB and 16KB, outperforming CacheLib slightly when objects sizes are 8KB. We were unable to pinpoint the cause of this trend. Nonetheless, CacheLib compares favorably to both NGINX and ATS across a wide range of object sizes.

LSM tree-based stores. It is natural to ask whether existing storage systems that target flash devices could be used as flash caching systems. In particular, Facebook's RocksDB [13] key-value store provides hybrid DRAM and flash storage by using a Log-Structured Merge-Tree (LSM Tree). We investigated whether RocksDB could be used as a hybrid look-aside cache for application data by deploying RocksDB in production to cache data from the *SocialGraph* workload.

RocksDB trades off higher space usage in favor of lower write and delete latencies, using tombstones to defer deletes operations until compaction time [13]. However, most compaction methods are computationally expensive and must be done sparingly. It is therefore infeasible to use RocksDB's `Delete` method to perform targeted evictions of objects, since compaction does not happen frequently enough for deletes to control the flash footprint of the cache. If RocksDB fills a flash device, it begins failing write and delete operations. This is particularly problematic in the *SocialGraph* system, which relies on deletes to maintain cache consistency. If a *SocialGraph* cache fails a certain number of deletes, the policy is to perform a cold restart (see Figure 15) to restore consistency.

As an alternative, we tested RocksDB using FIFO compaction, which simply evicts the oldest data when the size of the store exceeds its desired size. This compaction method is lightweight enough to run constantly and effectively limit RocksDB's flash usage. Evicting the oldest data will tend to evict the least recently *updated* objects, but these are generally not the same as the least recently *used* objects. RocksDB does not provide facilities for tracking which blocks contain recently used objects. Due to its simple eviction policy, RocksDB achieved only a 53% hit ratio compared to CacheLib's 76% hit ratio when tested with a production *SocialGraph* workload. Additionally, RocksDB under FIFO compaction suffers from severe read amplification and thus required 50% higher CPU utilization than CacheLib in order to meet production throughput levels. Hence, although some of the principles of LSM tree-based solutions can be carried over to the design of flash caches, we conclude that RocksDB itself is not suitable for caching at Facebook.

5.2. Characterizing CacheLib in Production

We quantify the impact that CacheLib has had on the Facebook production environment by considering the notable caching improvements that CacheLib has introduced.

DRAM overhead. By design, the DRAM overheads of the LOC and SOC are small; in production we measure less than 0.1% and 0.2%, respectively. The DRAM Cache has generally low ($< 7\%$) overhead. There are two main sources of DRAM overhead: slab class fragmentation and metadata overhead (Section 4.2). Tuning CacheLib's slab classes is crucial to limit fragmentation. Tuning currently happens manually. Without tuning, fragmentation overhead would more than double in many clusters. Unfortunately, we are not aware of automated tuning algorithms for slab-class boundaries[2]. A detailed analysis of DRAM overhead appears in Appendix B.

[2]Prior work has considered how to partition cache space between fixed slab classes [22] but not how to optimally define boundaries in the first place. Conceptually, this problem resembles the facility placement problem on a line [15], but we are not aware of optimal algorithms.

Flash endurance. CacheLib is designed to limit the rate of writes to flash in order to prolong flash device lifetimes (see Section 4.2). We now evaluate the effectiveness of this design.

The LOC incurs application-level write amplification due to fragmentation from the use of 4KB pages and size classes. Fragmentation is generally small, but *Storage* and *CDN* caches have 4.9% and 7% fragmentation overhead, respectively. To further reduce write amplification, CacheLib has recently introduced a new feature which buffers all writes to a region before flushing it to disk. This allows the application to write in sizes aligned to as low as 512 bytes, reducing fragmentation in *CDN* caches from 7% to 2%.

The LOC's use of FIFO eviction instead of LRU allows CacheLib to write to flash sequentially. Writing sequentially reduced device-level write amplification from 1.5× to 1.05× at the expense of slight increase in application-level write amplification. The net effect was a 15% reduction in the number of NAND writes to the flash device per second.

The SOC incurs application-level write amplification due to always writing 4KB (even as object sizes $<$ 2KB). On average, we measure this to be around 6.5× the number of inserted bytes. The SOC also incurs significant device-level write amplification from writing random 4KB pages [43]. We measure this overhead to be between between 1.1× (for *Lookaside*) and 1.4× (for *Storage*) depending on the workload.

To achieve these levels of device-level write amplification, flash is typically overprovisioned by 50%. This overprovisioning is offset by the space efficiency of the SOC and the low cost of flash relative to DRAM, but reducing flash overprovisioning while maintaining the current level of performance is an open challenge at Facebook.

To further limit the number of bytes written to a flash device, CacheLib uses admission policies for flash caches. The default CacheLib admission policy, which admits objects with a fixed probability, prolongs flash device lifetimes, but also decreases hit ratios by rejecting objects at random. CacheLib also includes *reject first* admission policies, which reject objects the first *n* times they are evicted from DRAM.

Recently, CacheLib was updated to include a more advanced admission policy, similar to the *Flashield* policy proposed in [32], which makes flash admission decisions by trying to predict an object's future popularity. Flashield's predictions are based on how many hits an object receives while in DRAM. At Facebook, however, many objects do not stay in DRAM long enough to get multiple hits. Thus, CacheLib implements efficient tracking of object request frequencies beyond their DRAM-cache lifetimes. These frequencies are then used to predict how many hits an object would receive if admitted to flash. Our experience with this advanced admission policy is described in detail in Appendix C. The advanced admission policy reduced the rate of writes to flash by 44% in *SocialGraph* without decreasing hit ratios.

Hit ratios. CacheLib's DRAM cache initially used a variant of the LRU eviction policy. A notable improvement in hit ratios across systems occurred when we deployed a 2Q-based eviction policy [54]. For example, the hit ratio for *SocialGraph* caches increased by 5 percentage points and the hit ratio for *CDN* caches increased by 9 percentage points.

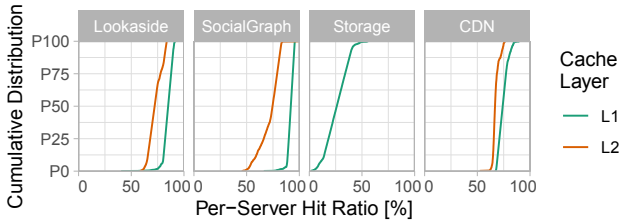


Figure 14: *Distribution of hit ratios for servers in the top four CacheLib users during a typical day.*

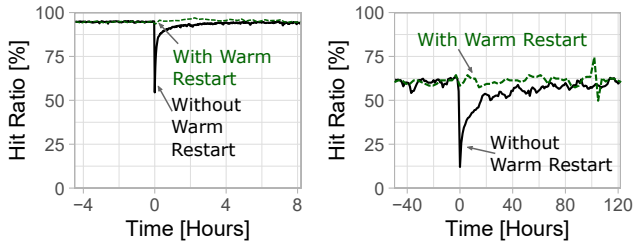


Figure 15: *SocialGraph L1 cache (left) and L2 cache (right) hit ratios during a cache restart. Hit ratios suffer when warm restarts are disabled.*

An even larger improvement in hit ratios resulted from the deployment of high-capacity hybrid DRAM-flash caches. Services requiring massive cache capacities generally consist of a two-layer hierarchy where "L1" DRAM-only cache forward misses to "L2" hybrid DRAM-flash caches. To see the improvement due to hybrid caches, we compare *SocialGraph*'s L2 caches from a deployment which uses hybrid caches to *SocialGraph*'s L2 caches from a deployment which still uses DRAM-only caches. The DRAM-only L2 caches for *SocialGraph* currently achieve a 25% hit ratio. The hybrid-cache L2 offers 20× more cache capacity, achieves a 60% hit ratio, and costs 25% less than the DRAM-only deployment.

Figure 14 shows hit ratio distributions for L1 and L2 caches for *Lookaside*, *SocialGraph*, and *CDN* clusters, some of the largest CacheLib deployments. L1 caches achieve much higher hit ratios than L2 caches, with median hit ratios ranging from 75% (*CDN*) to 92% (*SocialGraph*) while median L2 cache hit ratios range from 67% (*CDN*) to 75% (*SocialGraph*). The combined hit ratios of these systems are very high: ranging between 95-99%.

Impact of warm restarts. Figure 15 shows the hit ratios of L1 and L2 *SocialGraph* caches restarting without performing a warm restart. Without this feature enabled, a cache restart causes a dip in hit ratio, which slowly returns to normal. This is particularly damaging in L2 hybrid caches where large-capacity caches can take several days to "warm-up". Such a hit ratio dip can translate into temporary overload on backend systems, which assume a relatively stable arrival rate.

6. Experience and Discussion

Facebook's experience with CacheLib reveals a great deal about the trajectory of modern caching systems.

New features are adopted by many systems. One might expect that many CacheLib features end up being suitable for only a small number of services. However, our experience shows a trend in the opposite direction: *features developed for one particular service are frequently adopted by many other CacheLib-based services.* For example, hybrid caches and efficient object expirations (TTLs), were both added after the initial deployment of CacheLib. Today, hybrid caches are used by five large CacheLib use cases. Object expirations were originally added to enforce fair sharing in look-aside caches, but were later adopted by CDN caches, which need to refresh static content periodically. Nevertheless, not every feature is used by every system. Using a general-purpose caching engine is not equivalent to developing a single, one-size-fits-all approach to caching. Instead, we aim to benefit from extracting common caching functionality while still allowing a high degree of flexibility for cache customization.

Performance improvements help many systems. Even small performance improvements in CacheLib (see Section 5.2) have an outsized impact due to the broad deployment of CacheLib-based systems at Facebook. Deploying new features typically involves a simple code update or configuration change. The ability to make centralized improvements motivates a continuous effort to optimize the CacheLib code base. For example, while writing this paper, the LOC index implementation (see Section 4) changed to use a new sparse hashmap implementation, lowering CPU utilization by 0.5% with no change in memory overhead. While a 0.5% CPU decrease in a single system may not be worth the development cost, a 0.5% decrease across all of Facebook's hybrid caches amounts to a massive resource savings. This highlights the advantage of a common engine over specialization.

Improved stability. Another benefit of a common caching engine is improved stability due to the reduction of previously disjoint implementations to a single mature, well-tested platform. As new systems are built, using CacheLib greatly reduces the number of new lines of code that must be introduced into the production environment. This reduces the potential for production incidents and outages. CacheLib also provides explicit mechanisms for stability informed by years of experience deploying caches in the production environment.

No single caching system dominates. One can ask whether it might be sufficient to focus CacheLib engineering efforts on accommodating a small set of use cases instead of deploying CacheLib widely. To answer this question, we compare the total amounts of DRAM cache used by each system[3]. Figure 16 shows that the top ten users account for 89% of all DRAM cache usage, but no single service dominates. For example, the top two services account for only 25% and 20% of DRAM usage, respectively. Hence, unless CacheLib can accommodate many diverse use cases, the overall gains from optimizing CacheLib would be limited.

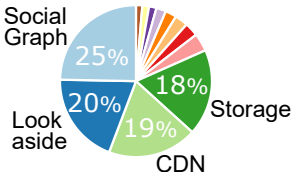


Figure 16: *A wide range of Facebook services are built using CacheLib. We measure a service's deployment size in terms of its total DRAM cache size. No service has more than 25% of the total cache space across Facebook services.*

Flash caching signals a paradigm shift. One might think that cache hit ratios are generally high, and hence expect little benefit from the additional cache capacity afforded by flash. While this is true in some cases, high hit ratios do not always imply that additional cache capacity is futile. Specifically, engineers provision caches to equalize the marginal cost of the next byte of DRAM with the marginal benefit of the ensuing increase in hit ratio. Flash caches alter this cost calculation, lowering the marginal cost of additional cache capacity by an order of magnitude. This makes it worthwhile to not only increase cache capacities dramatically, but to deploy new hybrid caches that did not make sense with DRAM alone.

Additionally, the benefit of a cache hit is no longer strictly a latency proposition for most systems. While a classical view of caching suggests that caching is only worthwhile if it reduces average memory access time [46], this ignores the knock-on effects of a cache miss such as increased network congestion, backend load, and backend power usage. From this perspective, a cache hit in flash is as valuable as a DRAM hit, even though flash is several orders-of-magnitude slower than DRAM. This again tips the scales of the marginal cost calculation in favor of deploying flash caches.

CacheLib does not always lead to performance gains. CacheLib-based systems have not always outperformed the specialized systems they replaced from the outset. For example, the first CacheLib-based implementation of the *CDN* system was not able to match the performance of the original *CDN* system, which optimized for flash caching by implementing advanced eviction policies with low flash write rates. The first CacheLib-based implementation of *CDN* achieved a 10% lower hit ratio and 20% higher flash write rate than the specialized system in testing.

Before the CacheLib-based implementation of *CDN* was deployed, optimizations were added to CacheLib to improve the hybrid caching mechanism. The LOC eviction policy was expanded from pure FIFO eviction to include a readmission policy which can readmit frequently requested objects when they are evicted. Write buffers were also added between the DRAM and flash caches. These buffers reduce application-level write amplification by reducing the internal fragmentation due to 4KB aligned writes. The write buffers also allow CacheLib to issue fewer, larger writes to flash, which reduces device-level write amplification.

[3]Not all services use hybrid caches, especially throughput-focused L1 caches.

The improved LOC eviction policy achieved a hit ratio close to that of the specialized system while performing 10% fewer writes to flash than the specialized system. Both of these optimizations add almost no overhead if turned off, and ended up improving the performance of other CacheLib-based systems as well. *Lookaside*, for example, saw a 25% reduction in P99 flash read latency, and a 2% reduction in flash write rate after these changes.

The *CDN* example illustrates the common case in balancing the generalization-versus-specialization tradeoff: CacheLib does not always address the needs of every use case from the outset. However, the features needed by specialized systems are often not fundamentally incompatible with the design of CacheLib. If one is willing to invest time into building the necessary features into CacheLib, they will gain access to CacheLib's full feature set while exporting new optimizations to the rest of the Facebook's caching systems.

CacheLib does not work for every use case. Although CacheLib handles many use cases, we are aware of limitations that have prevented some from adopting CacheLib. For instance, some ad-serving systems rely on caching *nested* data structures. In order to control its memory usage and quickly serialize `Items` from DRAM into flash, CacheLib only supports data structures that map into a flat address space. These ad-serving systems were thus unable to adopt CacheLib.

Another example is RocksDB, which wanted to use CacheLib to implement its internal page buffer. CacheLib's C++ API leverages object constructors and destructors to perform reference counting for `ItemHandle` objects. This ultimately prevented programmers from integrating CacheLib with RocksDB's C-style code base. However, the ease of automatic reference counting has led to widespread adoption of CacheLib for C++- and Rust-based use cases.

7. Related Work

There is vast body of research on caching systems including in-depth descriptions of individual production caches. We review prior work from industry and academia relevant in the context of web and data center caches.

Production caching systems. Caching systems are found within many major web services. Akamai's geo-distributed CDN [9, 28, 39, 67, 81, 85], Microsoft's web caching systems [8], Amazon's use of aggregation caches [26], and Facebook's many individual caching systems [5,48,71,72,86] are all documented in the literature. Similarly, Twitter [42, 92] and Reddit [33, 89] frequently talk about their DRAM caches based on open-source caching systems. CacheLib addresses a superset of the challenges faced by these individual systems, providing a single, general-purpose caching engine.

Research caching systems. Academic research has considered optimizing many different aspects of caching systems. These include building highly concurrent systems [6, 9, 35, 62, 64] and improving hit ratios [6, 9, 10, 21, 50, 51, 61, 88]. Facebook's goal is to use CacheLib as a platform to more easily evaluate and deploy systems based on this research.

While the literature mainly focuses on DRAM caching, there is some prior work on flash caching [32, 57, 60, 86]. CacheLib incorporates ideas from [86] and [60] to reduce write amplification by doing FIFO eviction on flash. Likewise, CacheLib includes the admission policy of [57] and a variant of the admission policy from [32] (see Appendix C).

Although dynamic cache partitioning is possible in CacheLib, the impact of existing research on cache partitioning policies is limited at Facebook. Partitioning can be used to eliminate performance cliffs in a cache's hit ratio as a function of size [7, 22, 88], but performance cliffs are not a major issue at Facebook. As the authors of RobinHood [8] note in their work, the RobinHood partitioning scheme is limited when infrastructure is shared between different backend systems, which is the case at Facebook. Additionally, the computational overhead of retrieving the size of objects stored on flash is too high to use size-aware sharding [27] in practice.

8. Conclusions

Caching is an important component of modern data-center applications, and this paper has only scratched the surface of its many challenges and opportunities. CacheLib shows that it is feasible to build a general-purpose caching engine to address a wide variety of caching use cases. In sharing our experience of building and deploying CacheLib, we hope to solicit ideas and feedback from the growing community of caching researchers and to encourage other production systems to share their architectures and workloads. We hope that CacheLib will serve as an aggregation point for best practices in industry and academia, enabling a continual improvement in performance and facilitating the deployment of caches in many new applications. There are many exciting directions to explore in future caching systems, including *(i)* better resource-management policies (e.g., eviction/admission policies, memory management); *(ii)* emerging hardware platforms (e.g., FPGA acceleration, non-volatile memories, zoned-namespace SSDs); and *(iii)* novel application features (e.g., as seen in negative caching). We look forward to growing CacheLib to address these challenges and many others.

Appendix

A. Cache Lookup Order and Latency

CacheLib uses the lookup sequence 1) DRAM cache, 2) LOC, 3) SOC. Note that an object's size is not known in advance. So, after a DRAM cache miss, CacheLib does not know whether the object is stored in the LOC or the SOC. Thus, it has to query one of them first, and on a miss, query the other.

The order for CacheLib's lookup order is motivated by the following analysis of average lookup penalties (also known as average memory access time, AMAT [46]). We consider the lookup penalty for each cache component as the time to determine that an object is not cached in this component. Our key assumption is that reading from DRAM is orders of magnitude faster than flash reads (e.g., 100ns compared to

16us [25]). Thus, the lookup penalty for the DRAM cache is a few memory references (say 500ns).

To calculate the penalty for the LOC, recall that the LOC stores neither an object's key nor the object's exact size in memory to reduce DRAM metadata overhead. The LOC is indexed via 4-byte hash-partitioned B-trees, which each use 4-byte hashes to identify an object's offset. If the overall 8-byte-hash does not have a hash collision, then the LOC's lookup penalty constitutes a few memory references (say 1us, due to hash operations). If there is a hash collision, the LOC requires a flash read (16us) to compare the object key and determine the miss status. Assuming the smallest LOC object size (2KB) and 1TB of flash, at most 536 million objects are stored in the LOC. Thus, the probability of an 8-byte-hash collision can be calculated to be less than one in a million and the LOC's average lookup penalty is slightly more than 1us.

To calculate the penalty for the SOC, recall that the SOC does not use an in-memory index. The SOC uses a per-page Bloom filter (say 1us) to opportunistically determine the miss status. However, as these Bloom filters are small, their error rate is 10%. In case of a Bloom filter error, the SOC requires a flash read (16us) to compare the object key. The SOC's average lookup penalty is thus 2.6us.

The average latency (AMAT) of CacheLib with the default order (1) DRAM cache, (2) LOC, (3) SOC is as follows, where L denotes lookup latency and H hit ratio: $L(\text{DRAM}) + \big(1 - H(\text{DRAM})\big) \times \Big(L(\text{LOC}) + \big(1 - H(\text{LOC})\big) \times L(\text{SOC})\Big)$. With the order of SOC and LOC inverted, the average latency would increase by several microseconds, depending on the LOC and SOC hit ratios. Thus CacheLib queries the SOC last.

B. Details on DRAM Overheads

DRAM Cache. We measure CacheLib's DRAM cache overhead as the ratio between its total memory footprint and the sum of cached key and value sizes. We further break up overheads into slab-class fragmentation and metadata. Across *Lookaside*, *Storage*, and *SocialGraph*, we find that overall overheads are between 2.6 and 7% and evenly divided between fragmentation and metadata.

	Lookaside	*Storage*	*SocialGraph*
Fragmentation	3.9%	3%	1.6%
Metadata	3%	4%	1%
Overall overhead	6.9%	7%	2.6%

Large Object Cache. Recall that, while the LOC uses an 8-byte hash, 4-bytes are used to partition B-tree and thus do not need to be counted. So, the LOC stores 4-bytes for key hashes, 4-bytes for flash offsets, and an average of 2.5-bytes per item for B-tree pointers. For the small LOC object, this is 0.61%. In production systems, this overhead is low and ranges from to 0.01% (*Storage*) to 0.1% (*Lookaside*).

C. Advanced Admission Policies for Flash

One significant challenge in using flash for caching is respecting the limited write endurance of flash devices. If all DRAM evictions in a hybrid cache were admitted to flash, we would observe write rates 50% above the rate which allows flash devices to achieve their target life span. A flash admission policy thus plays an important role in CacheLib's performance.

Flashield [32] is a recently proposed flash admission policy. Flashield relies on observing an object as it traverses the DRAM portion of a hybrid cache. When an object is evicted from DRAM, Flashield makes a flash admission decision based on how often the object was accessed while in DRAM.

Unfortunately, DRAM lifetimes at Facebook are too short for Flashield to be effective. A significant number of objects are popular enough to produce hits if stored on flash, but do not receive DRAM cache hits. In fact, for an L2 *Lookaside* cache, only 14% of objects being considered for flash admission have received either a read or a write while in DRAM.

To adapt the main idea behind Flashield to Facebook's environment, CacheLib explicitly gathers features about objects beyond their DRAM-cache lifetime. We use Bloom filters to record the past six hours of accesses[4]. Additionally, we change the admission policy's prediction metrics from the abstract notion of "flashiness" to instead directly predict the number of reads an object is expected to receive in the future.

Our advanced admission policy was trained and deployed in production for *SocialGraph*. The default admission policy for CacheLib flash caches is to admit objects with a fixed probability that keeps flash write rates below a target rate in expectation. Compared to this default admission policy, the advanced admission policy wrote 44% fewer bytes to the flash device without decreasing the cache hit ratio. Hence, while training the models required for the advanced admission policy can be cumbersome, this policy gain significantly extend the lifetime of flash devices in production.

Acknowledgements

This work is supported by NSF-CMMI-1938909, NSF-CSR-1763701, NSF-XPS-1629444, a 2020 Google Faculty Research Award, and a Facebook Graduate Fellowship. We also thank the members and companies of the PDL Consortium (Alibaba, Amazon, Datrium, Facebook, Google, HPE, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, Two Sigma and Western Digital) and VMware for their interest, insights, feedback, and support.

[4] Specifically, we are using 6 Bloom filters, each set to hold 1 hour of accesses. Each hour, the oldest Bloom filter is reset and used to track the upcoming hour. These Bloom filters are configured for 0.02% false positives at maximum observed query rate. The space efficiency of Bloom filters is necessary to avoid using up too much DRAM - using 8 bytes per stored key to store history would be too much space overhead.

References

[1] Companies using apache traffic server. https://trafficserver.apache.org/users.html. Accessed: 2019-04-22.

[2] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: diving into data at facebook. *VLDB*, 6(11):1057–1067, 2013.

[3] Apache. Traffic Server, 2019. Available at https://trafficserver.apache.org/, accessed 04/13/19.

[4] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *ACM SIGMOD*, pages 1185–1196, 2013.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, volume 40, pages 53–64, 2012.

[6] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd: Improving hit rate by maximizing hit density. In *USENIX NSDI.*, pages 1–14, 2018.

[7] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *IEEE HPCA.*, pages 64–75, 2015.

[8] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-Balter, and Siddhartha Sen. RobinHood: Tail latency-aware caching - dynamically reallocating from cache-rich to cache-poor. In *USENIX OSDI*, 2018.

[9] Daniel S. Berger, Ramesh Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a cdn. In *USENIX NSDI*, pages 483–498, March 2017.

[10] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, pages 499–511, 2017.

[11] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[12] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695, 2006.

[13] Dhruba Borthakur. Under the hood: Building and open-sourcing rocksdb, 2013. Facebook Engineering Notes, available at http://bit.ly/2m02DGV, accessed 09/02/19.

[14] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE INFOCOM*, pages 126–134, 1999.

[15] Jack Brimberg and Abraham Mehrez. Location and sizing of facilities on a line. *Top*, 9(2):271–280, 2001.

[16] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. GAIA: An OS page cache for heterogeneous systems. In *USENIX ATC*, pages 661–674, 2019.

[17] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, 2013.

[18] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *ACM SIGPLAN Notices*, volume 33, pages 139–149, 1998.

[19] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *ACM SIGMOD*, pages 275–290, 2018.

[20] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *ACM SIGMETRICS*, pages 145–156, 2005.

[21] Ludmila Cherkasova. Improving WWW proxies performance with greedy-dual-size-frequency caching policy. Technical report, Hewlett-Packard Laboratories, 1998.

[22] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, pages 379–392, 2016.

[23] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[24] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *ACM SoCC*, pages 143–154, 2010.

[25] Jeff Dean and P Norvig. Latency numbers every programmer should know, 2012.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SOSP*, volume 41, pages 205–220, 2007.

[27] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *USENIX NSDI*, pages 79–94, 2019.

[28] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, and William E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[29] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., November 2007.

[30] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *ACM Middleware*, pages 94–106, 2018.

[31] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *IEEE Euromicro PDP*, pages 146–153, 2014.

[32] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *USENIX NSDI*, pages 65–78, 2019.

[33] Daniel Ellis. Caching at reddit, January 2017. Available at `https://redditblog.com/2017/01/17/caching-at-reddit/`, accessed 09/02/19.

[34] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. 7 edition, 2015.

[35] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, pages 371–384, 2013.

[36] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SoCC*, page 23, 2011.

[37] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[38] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *ACM IMC*, pages 15–28, 2007.

[39] David Gillman, Yin Lin, Bruce Maggs, and Ramesh K Sitaraman. Protecting websites from attack with secure delivery networks. *IEEE Computer*, 48(4):26–34, 2015.

[40] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. The stretched exponential distribution of internet media access patterns. In *ACM PODC*, pages 283–294, 2008.

[41] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. Trade-offs in optimizing the cache deployments of cdns. In *IEEE INFOCOM*, pages 460–468, 2014.

[42] Mazdak Hashemi. The infrastructure behind twitter: Scale, January 2017. Available at `https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html`, accessed 09/02/19.

[43] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The unwritten contract of solid state drives. In *ACM EuroSys*, pages 127–144, 2017.

[44] Leif Hedstrom. Deploying apache traffic server, 2011. Oscon.

[45] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010.

[46] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 4 edition, 2011.

[47] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[48] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *ACM SOSP*, pages 167–181, 2013.

[49] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage*, 12(2):1–24, 2016.

[50] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *ACM/IEEE ISCA*, pages 78–89, 2016.

[51] Jaeheon Jeong and Michel Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, 2006.

[52] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *USENIX FAST*, volume 4, pages 8–8, 2005.

[53] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, pages 121–136, 2017.

[54] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[55] Poul-Henning Kamp. Varnish notes from the architect, 2006. Available at `https://www.varnish-cache.org/docs/trunk/phk/notes.html`, accessed 09/12/16.

[56] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *ACM/IEEE ISCA*, pages 158–169, 2015.

[57] Eunji Lee and Hyokyung Bahn. Preventing fast wear-out of flash cache with an admission control policy. *Journal of Semiconductor technology and science*, 15(5):546–553, 2015.

[58] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *ACM SOSP*, pages 447–461, 2019.

[59] Jacob Leverich. The mutilate memcached load generator, August 2012. Available at `https://github.com/leverich/mutilate`, accessed 08/20/19.

[60] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage*, 13(3):24, 2017.

[61] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *EUROSYS*, pages 1–15, 2015.

[62] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM ISCA*, pages 476–488, 2015.

[63] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.

[64] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI*, pages 429–444, 2014.

[65] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *ACM ASPLOS*, pages 795–809, 2017.

[66] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST*, pages 143–157, 2019.

[67] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR*, 45:52–66, 2015.

[68] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *ACM EuroSys*, pages 183–196, 2012.

[69] Tony Mauro. Why netflix chose nginx as the heart of its cdn. `https://www.nginx.com/blog/why-netflix-chose-nginx-as-the-heart-of-its-cdn`. Accessed: 2020-04-22.

[70] Domas Mituzas. Flashcache at facebook: From 2010 to 2013 and beyond, October 2013. Facebook Engineering, available at `https://bit.ly/3cMXfvT`, accessed 04/23/20.

[71] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm BLOB storage system. In *USENIX OSDI*, pages 383–398, 2014.

[72] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *USENIX NSDI*, pages 385–398, 2013.

[73] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: core-aware thread management. In *USENIX OSDI*, pages 145–160, 2018.

[74] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *USENIX OSDI*, pages 401–417, 2016.

[75] Redis, 2019. `https://redis.io/`, accessed 04/23/20.

[76] Kay A. Robbins and Steven Robbins. *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading*. Prentice-Hall, 2003.

[77] Emanuele Rocca. Running Wikipedia.org, June 2016. Available at `https://www.mediawiki.org/`

wiki/File:WMF_Traffic_Varnishcon_2016.pdf, accessed 09/12/16.

[78] Goldwyn Rodrigues. Taming the oom killer. *LWN*, February 2009.

[79] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *ACM EuroSys*, page 267–280, 2012.

[80] Chris Sears. The elements of cache programming style. In *USENIX ALS*, pages 283–298, October 2000.

[81] Ramesh K. Sitaraman, Mangesh Kasbekar, Woody Lichtenstein, and Manish Jain. Overlay networks: An Akamai perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons, 2014.

[82] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *USENIX NSDI*, pages 529–544, 2020.

[83] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *ACM IMC*, pages 41–54, 2004.

[84] Akshitha Sriraman and Thomas F Wenisch. μtune: Auto-tuned threading for OLDI microservices. In *USENIX OSDI*, pages 177–194, 2018.

[85] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *ACM CoNEXT*, pages 55–67, 2017.

[86] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: advanced photo caching on flash for facebook. In *USENIX FAST*, pages 373–386, 2015.

[87] Francisco Velázquez, Kristian Lyngstøl, Tollef Fog Heen, and Jérôme Renard. *The Varnish Book for Varnish 4.0*. Varnish Software AS, March 2016.

[88] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *USENIX*, pages 487–498), 2017.

[89] Neil Williams. Reddit's architecture, November 2017. QCon SF slide set, available at https://qconsf.com/sf2017/system/files/presentation-slides/qconsf-20171113-reddits-architecture.pdf, accessed 09/02/19.

[90] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. Nvm-cached: An nvm-based key-value cache. In *ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 1–7, 2016.

[91] Yuejian Xie and Gabriel H Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News*, 37(3):174–183, 2009.

[92] Yao Yue. Cache à la carte: a framework for in-memory caching, September 2015. Strange Loop slide set, available at https://www.youtube.com/watch?v=pLRztKYvMLk, accessed 09/02/19.

[93] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC*, pages 91–104, 2001.

[94] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A Kozuch. Saving cash by using less cache. In *USENIX HOTCLOUD*, 2012.

Twine: A Unified Cluster Management System for Shared Infrastructure

Chunqiang Tang Kenny Yu Kaushik Veeraraghavan Jonathan Kaldor
Scott Michelson Thawan Kooburat Aravind Anbudurai Matthew Clark Kabir Gogia
Long Cheng Ben Christensen Alex Gartrell Maxim Khutornenko Sachin Kulkarni
Marcin Pawlowski Tuomas Pelkonen Andre Rodrigues Rounak Tibrewal
Vaishnavi Venkatesan Peter Zhang

Facebook Inc.

Abstract

We present *Twine*, Facebook's cluster management system which has been running in production for the past decade. Twine has helped convert our infrastructure from a collection of siloed pools of customized machines dedicated to individual workloads, into a large-scale shared infrastructure with fungible hardware.

Our goal of ubiquitous shared infrastructure leads us to some decisions counter to common practices. For instance, rather than deploying an isolated control plane per cluster, Twine scales a single control plane to manage one million machines across all data centers in a geographic region and transparently move jobs across clusters.

Twine accommodates workload-specific customization in shared infrastructure, and this approach further departs from common practices. The *TaskControl* API allows an application to collaborate with Twine to handle container lifecycle events, e.g., restarting a ZooKeeper deployment's followers first and its leader last during a rolling upgrade. *Host profiles* capture hardware and OS settings that workloads can tune to improve performance and reliability; Twine dynamically allocates machines to workloads and switches host profiles accordingly.

Finally, going against the conventional wisdom of prioritizing stacking workloads on big machines to increase utilization, we universally deploy power-efficient small machines outfit with a single CPU and 64GB RAM to achieve higher performance per watt, and we leverage autoscaling to improve machine utilization.

We describe the design of Twine and share our experience in migrating Facebook's workloads onto shared infrastructure.

1 Introduction

The advent of computation as a utility has led organizations to consolidate their workloads onto *shared infrastructure*, a common pool of resources to run any workload. Cluster management systems help organizations utilize shared infrastructure effectively through automation, standardization, and economies of scale. Cluster management systems have made large progress in the past decade, from Mesos [17], Borg [39], to Kubernetes [23]. Existing systems, however, still have limitations in supporting large-scale shared infrastructure:

1. They usually focus on isolated clusters, with limited support for cross-cluster management as an afterthought. These silos may strand unused capacity in clusters.
2. They rarely consult an application about its lifecycle management operations, making it more difficult for the application to uphold its availability. For example, they may unknowingly restart an application before it has built another data replica, rendering the data unavailable.
3. They rarely allow an application to provide its preferred custom hardware and OS settings to shared machines. Lack of customization may negatively impact application performance on shared infrastructure.
4. They usually prefer big machines with more CPUs and memory in order to stack workloads and increase utilization. If not managed well, underutilized big machines waste power, often a constrained resource in data centers.

These limitations can lead to underdelivery of the promise of shared infrastructure: (1) artificially caps the sharing scope to one cluster; (2) & (3) highlight the tension between shared infrastructure's preference for standardization and applications' needs for customization; (4) calls for a shift of focus from single-machine utilization to global optimization.

In this paper, we describe how we address the above limitations in *Twine*, Facebook's cluster management system. Our two insights are 1) we scale a single Twine control plane to manage one million machines across data centers in a geographic region while providing high reliability and performance guarantees, and 2) we support workload-specific customization, which allows applications to run on shared infrastructure without sacrificing performance or capabilities.

Twine packages applications into Linux containers and manages the lifecycle of machines, containers, and applications. A *task* is one instance of an application deployed in a container, and a *job* is a group of tasks of the same application.

A single control plane to manage one million machines. A region consists of multiple data centers, and a data center is usually divided into *clusters* of tens of thousands of machines connected by a high-bandwidth network. As with Borg [39] and Kubernetes [23], an isolated control plane per cluster results in stranded capacity and operational burden because workloads cannot easily move across clusters. For example, power-hungry jobs colocated in a cluster can trigger power capping [26, 41], affecting service throughput until humans move the problematic jobs to other clusters.

Similarly, large-scale hardware refresh in a cluster may result in idle machines and operational overhead. Our current hardware refresh granularity is 25% of a data center. Figure 1 shows the duration for all owners of thousands of jobs to migrate jobs out of a cluster prior to a hardware refresh in 2016. The P50 is at 7.5 days and the P100 is at 87 days. A large portion of the cluster sat idle in these ≈80 days while waiting for all jobs to migrate.

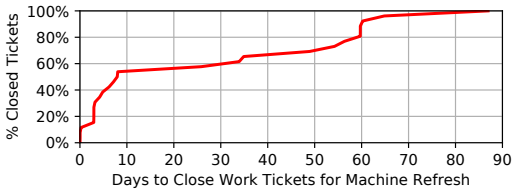


Figure 1: CDF of time to close job-migration work tickets.

To address the problems above, we scaled a single Twine control plane to manage one million machines across all data centers in a region. Unlike Kubernetes Federation [25], Twine scales out natively without an additional federation layer.

Collaborative lifecycle management. Cluster management systems generally lack visibility into how an application manages its internal state, leading to suboptimal handling of hardware and software lifecycle events that impact application availability. Figure 2 provides a stateful service example.

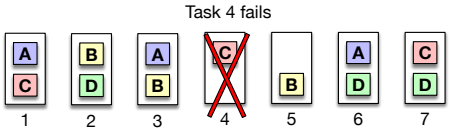


Figure 2: Replicas of data shards `A`–`D` are distributed across tasks 1-7. Tasks 1 and 3 should not be restarted concurrently for a software upgrade, as shard `A` would lose two replicas and become unavailable. If the machine hosting task 4 were to fail or be restarted for a kernel or firmware upgrade, the cluster management system would need to ensure that neither task 1 nor 7 is restarted concurrently in order to keep shard `C` available.

Twine provides a novel *TaskControl* API to allow applications to collaborate with Twine in handling task lifecycle events that impact availability. For example, an application may postpone a task restart and rebuild a lost data replica first.

Host-level customization. Hardware and OS settings may significantly impact application performance. For example, our web tier achieves 11% higher throughput by tuning OS settings. Twine leverages *entitlements*, our quota system, to handle hardware and OS tuning. For example, an entitlement for a business unit may allow it to use up to 30,000 machines. We associate each entitlement with a *host profile*, a set of host customizations that the entitlement owner can tune. Out of a shared machine pool, Twine dynamically allocates machines to entitlements and switches host profiles accordingly.

Power-efficient machines. Facebook's workloads have grown faster than our data center buildup. Power scarcity motivated us to maximize performance per watt, either by employing universal stacking on big machines or deploying power-efficient small machines. We found it challenging to stack large workloads on big machines effectively. Further, unlike a public cloud that needs to support diverse customer requirements, we only need to optimize for our internal workloads. These factors led to us to adopt small machines with a single CPU and 64GB RAM [32].

Shared infrastructure. As we evolved Twine to support large-scale shared infrastructure, we have been migrating our workloads onto a single shared compute pool, *twshared*, and a single shared storage pool. Twine supports both pools, but we focus on twshared in this paper. twshared hosts thousands of systems, including frontend, backend, ML, stream processing, and stateful services. While twshared does not host durable storage systems, it provides TBs of local flash to support stateful services that store state derived from durable storage systems. Figure 3 shows twshared's growth.

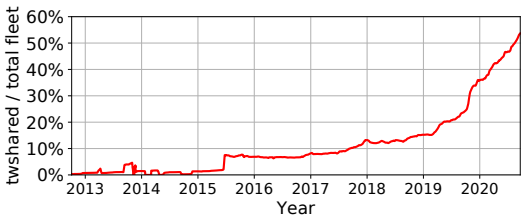


Figure 3: Growth of twshared. twshared was created in 2013, but adoption was limited in its first six years. We enhanced Twine and rebooted the adoption effort in 2018. twshared hosts 56% of our fleet as of October 2020, in contrast to 15% in January 2019. We expect that all compute services, ≈85% of our fleet, will run on twshared by early 2022, while the remaining 15% will run in a separate shared storage pool.

twshared has become our ubiquitous compute pool, as all new compute capacity lands only in twshared. We had broad conversations with colleagues in industry and are unaware of any large company that has achieved near 100% shared infrastructure consolidation.

The rest of the paper is organized as follow. §2 presents the design and implementation of Twine. §3 and §4 describe how we scale Twine to manage one million machines and do so reliably. §5 evaluates Twine. §6 shares our experience with driving twshared adoption. §7 describes lessons learned. §8 summarizes related work. Finally, §9 concludes the paper.

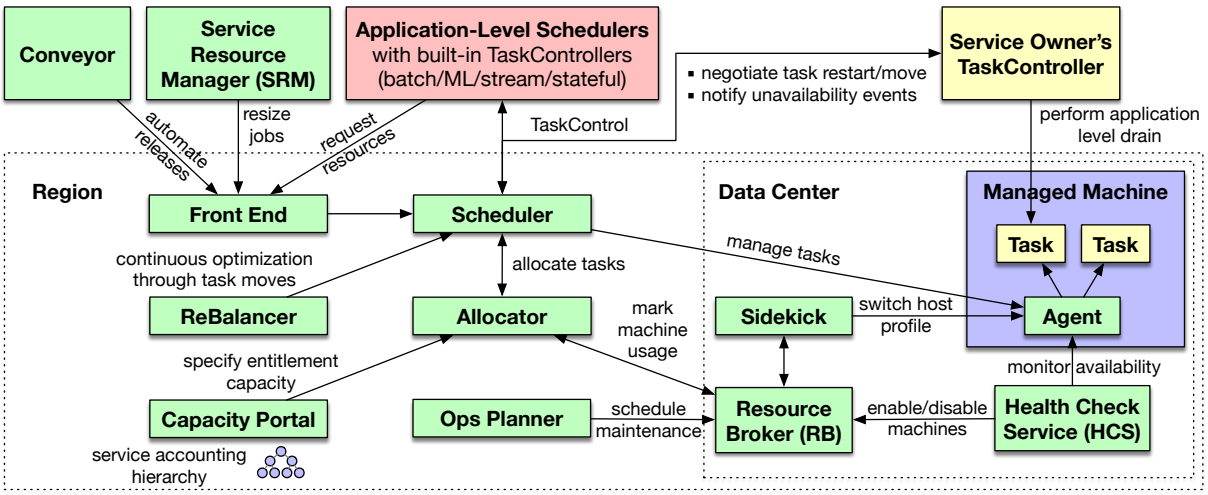


Figure 4: The Twine Ecosystem. Note a potential terminology confusion. The Twine *scheduler* corresponds to the Kubernetes [23] controllers, whereas the Twine allocator corresponds to the Kubernetes scheduler.

2 Twine Design and Implementation

Facebook currently operates out of 12 geo-distributed *regions*, with several more under construction. Each region consists of multiple *data center* (DC) buildings. A *main switchboard* (MSB) [41] is the largest fault domain in a DC with sufficient power and network isolation to fail independently. A DC consists of tens of MSBs each powering tens of rows that feed tens of racks of servers as shown in Figure 5.

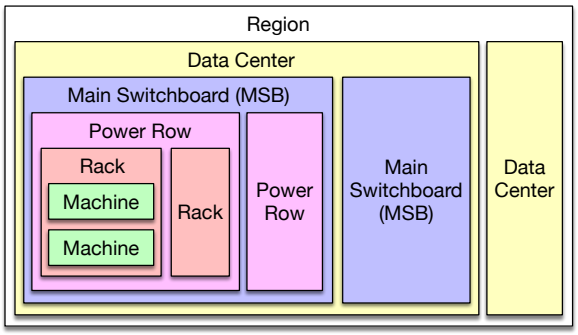


Figure 5: Data center topology.

Historically, a *cluster* was a subunit within a DC consisting of about ten thousand machines connected by a high-bandwidth network and managed by an isolated Twine control plane. Over time, our network transitioned to a fabric architecture [2, 14] that provides high bandwidth both within a DC and across DCs in a region, empowering a single Twine control plane to manage jobs across DCs.

2.1 Twine Ecosystem

Figure 4 shows an overview of Twine. The *Capacity Portal* allows users to request or modify *entitlements*, which associate capacity quotas with business units defined in the *service accounting hierarchy*. With a granted entitlement, a user deploys jobs through the *front end*. The *scheduler* manages job and task lifecycle, e.g., orchestrating a job's software release. If a job has a *TaskController*, the scheduler coordinates with the TaskController to make decisions, e.g., delaying a task restart to rebuild a lost data replica first. The *allocator* assigns machines to entitlements and assigns tasks to machines. *ReBalancer* runs asynchronously and continuously to improve the allocator's decisions, e.g., better balancing the utilization of CPU, power, and network. *Resource Broker* (RB) stores machine information and *unavailability events* that track hardware failures and planned maintenance. DC operators schedule planned maintenance through *Ops Planner*. The *Health Check Service* (HCS) monitors machines and updates their status in RB. The *agent* runs on every machine to manage tasks. *Sidekick* switches host profiles as needed. *Service Resource Manager* (SRM) autoscales jobs in response to load changes. *Conveyor* is our continuous delivery system.

2.2 Entitlements

Conceptually, an *entitlement* is a pseudo cluster that uses a set of dynamically allocated machines to host jobs. An entitlement grants a business unit a quota expressed as a count of machines of certain types (e.g., 2,000 Skylake machines) or as Relative Resource Units (RRU) akin to ECU in AWS.

A machine is either free or assigned to an entitlement, and it can be dynamically reassigned from one entitlement to another. An entitlement can consist of machines from different DCs in a region. Existing cluster management systems bind a job to a physical cluster. In contrast, Twine binds a job to an entitlement. Jobs in an entitlement stack with one another on machines assigned to the entitlement.

By default, Twine spreads tasks of the same job across DCs and MSBs as shown in Figure 6. This reduces buffer capacity

needed for fault tolerance [29]. Suppose a job's tasks are spread across 12 MSBs in one DC. We need $\frac{1}{12} \approx 8.3\%$ of buffer capacity to guard against the failure of one MSB. If the job's tasks are spread across five DCs' 60 MSBs, the needed buffer reduces to $\frac{1}{60} \approx 1.7\%$. For workloads that require better locality for compute and storage, Twine allows an entitlement to override the default spread policy and pin its machines and jobs to a specific DC. These workloads are in the minority.

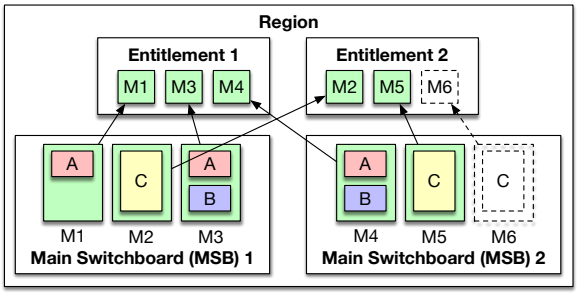


Figure 6: Entitlement example. `Entitlement 1` consists of machines `M1`, `M3`, and `M4` from different MSBs. Jobs `A` and `B` are bound to `Entitlement 1`, and job `C` is bound to `Entitlement 2`. Jobs `A` and `B` stack their tasks on machine `M3`. As job `C` grows, Twine adds machine `M6` to `Entitlement 2`.

The allocator assigns machines to entitlements, and it also assigns tasks to machines in an entitlement. For an entitlement with a quota of N machines, the number of machines actually assigned to the entitlement may vary between 0 and N, depending on the actual needs of jobs running in the entitlement. Figure 7 depicts an example of how an entitlement changes over time.

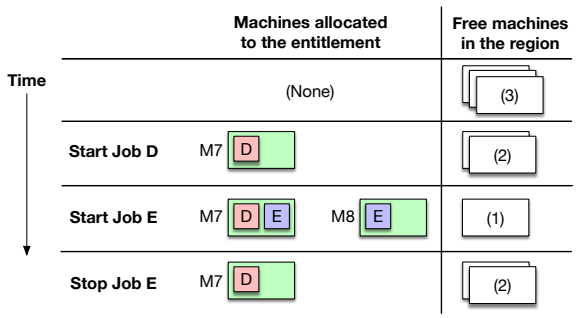


Figure 7: Allocation of machines and tasks. Initially, no machine is assigned to the entitlement. When job `D` starts, the allocator assigns machine `M7` to the entitlement. When job `E` starts, the allocator stacks one task on `M7` and adds machine `M8` to the entitlement to run `E`'s other task. When job `E` stops, the allocator returns `M8` to the free machine pool for use by other entitlements.

We optimized the allocator to make quick decisions when starting tasks; this optimization limits computation time and leads to best-effort outcomes. The addition or removal of machines and workload evolution may result in hotspots in CPU, power, or network. ReBalancer runs asynchronously and continuously to improve upon the allocator's allocation decisions by swapping machines across entitlements or moving tasks across machines. ReBalancer uses a constraint solver to perform these time-consuming global optimizations.

Entitlements help automate job movements across clusters. Consider a cluster-wide hardware refresh. We first add new machines from other clusters into the regional free machine pool (see the right side of Figure 7). Then the allocator moves tasks from machines undergoing hardware refresh to new machines acquired from the free machine pool, requiring no actions from the job owner. To migrate a task, Twine stops the task on the old machine and restarts it on the new machine. We do not use live container migration.

2.3 Allocator

One instance of Resource Broker (RB) is deployed to each DC. RB records whether a machine in the DC is free or assigned to an entitlement. A regional allocator fetches this information from all RBs in the same region, maintains an in-memory write-through cache, and subscribes to future changes.

The scheduler calls the allocator to perform a job allocation when a new job starts, an existing job changes size, or a machine fails. The allocation request contains an entitlement ID, an allocation policy, and a per-task map of which tasks need to be allocated or freed. The allocation policy includes hard requirements (e.g., using Skylake machines only) and soft preferences (e.g., spreading tasks across fault domains).

The allocator maintains an in-memory index of all machines and their properties to support hard requirement queries, such as "*all Skylake machines with available CPU $\geq$ 2RRU and available memory $\geq$ 5GB*." It needs to search machines beyond the ones already assigned to the entitlement because it may need to add more machines to the entitlement to host the job. After applying hard requirements, it applies soft preferences to sort the remaining machines.

A soft preference is expressed as a combination of 1) a machine property to partition machines into different bins with the same property value, and 2) a strategy to allocate tasks to these machine bins. For example, the allocator spreads tasks across fault domains by using a soft preference with `fault domain` as the machine property, and the strategy that assigns tasks evenly to the machine bins that represent fault domains.

The allocator uses multiple threads to perform concurrent allocations for different jobs, and relies on optimistic concurrency control to resolve conflicts. Before committing an allocation, a thread verifies that all impacted machines still have sufficient resources left for the allocation. If the verification fails, it retries a different allocation.

To avoid repeating the costly machine selection process, the allocator caches the allocation results at the job level. The allocator invalidates a cache entry if the job allocation request changes or the properties of the machines hosting the tasks change. The cache hit ratio is typically above 99%.

2.4 Scheduler

The scheduler manages the lifecycle of jobs and tasks. As the central orchestrator, the scheduler drives changes across

Twine components in response to different lifecycle events, including hardware failures, maintenance operations, power capping [41], kernel upgrades, job software releases, job resizing, task canary, and ReBalancer moving tasks.

The scheduler handles a machine failure as follows. When the Health Check Service detects a machine failure, it creates an unavailability event in Resource Broker, which notifies the allocator and scheduler. The scheduler disables the affected tasks in the service discovery system so that clients stop sending traffic to these tasks. A job is impacted by the machine failure if it has tasks running on the machine. If an impacted job has a TaskController, the scheduler informs the TaskController of the affected tasks. After the TaskController acknowledges that these tasks can be moved, the scheduler requests the allocator to deallocate the tasks and allocate new instances of the tasks on other machines. The scheduler instructs agents to start the new tasks accordingly. Finally, the scheduler enables the tasks in the service discovery system so that clients can send traffic to the newly started tasks.

The scheduler paces changes to a job's tasks to avoid application downtime. For example, regardless of reasons (e.g., hardware failure or software upgrade), if a job's total unavailable tasks exceed a user-configured threshold, no more tasks can be restarted for a software release. The scheduler has built-in support for commonly used lifecycle policies and offers the TaskControl API to implement more complex policies.

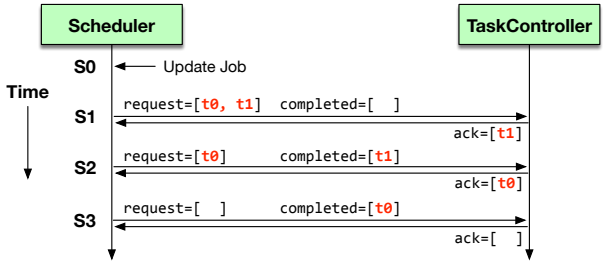
2.5 TaskControl

An application often knows best how to safely handle hardware or software lifecycle events that affect its availability, but it cannot inform the cluster management system how to orchestrate these actions. Figure 2 depicts one example. Another example is a ZooKeeper deployment that wishes to apply a software release to its followers first and its leader last [8]. Otherwise, an n-member ZooKeeper ensemble in the worst case experiences n leader failovers during a release. We designed the *TaskControl* API to allow applications to collaborate with Twine when deciding which task operations to proceed and which to postpone, as depicted in Figure 8.

Unlike software releases, maintenance events like a power device replacement cannot be blocked indefinitely by a TaskController; the scheduler gives the TaskController advance notices with a deadline to react. Upon reaching the deadline, the scheduler stops the remaining tasks on the affected machines, allowing maintenance to proceed. Before the deadline, a TaskController has multiple options: 1) move the tasks to other machines, 2) stop the tasks on the current machine and restart them after the maintenance completes, or 3) do nothing and keep the tasks running. For example, a top-of-rack switch maintenance typically incurs only a few minutes of network downtime, and a stateful service may prefer option 3 because rebuilding a data replica elsewhere takes longer than the maintenance itself.

```
service TaskController {
  TaskControlResponse process(TaskControlRequest request);
}
struct TaskControlRequest {
  string jobHandle;
  list<> request; // Pending task operations to be approved.
  list<> completed; // Completed task operations.
  list<> advanceNotices; // Upcoming planned maintenance.
  list<> allUnhealthyTasks; // Tasks unhealthy due to any reason.
  int sequenceNumber; // Increase after each call.
}
struct TaskControlResponse {
  list<> ack; // Approved task operations.
}
```

(a) TaskControl API.



(b) Calling sequence of the TaskControl API when handling a job update. The job has two tasks: `t0` and `t1`. At time `S0`, the user initiates a job update. At time `S1`, the scheduler requests the approval of updates on tasks `t0` and `t1`, with `request=[t0,t1]`. The application's TaskController can selectively approve updates for any subset of tasks in any order. It approves the update on task `t1` by replying `ack=[t1]`, but delays the update on task `t0` to keep one task available. At time `S2`, the scheduler completes the update on task `t1` with `completed=[t1]`, and requests an update on the remaining task `t0`. This time, the TaskController approves the request.

Figure 8: TaskControl API and an example of the calling sequence.

2.6 Host Profiles

Our fleet runs thousands of different services, and Figure 9 shows that the 50 largest services consume ≈70% of all capacity. Similar capacity skew exists in Borg as well [36].

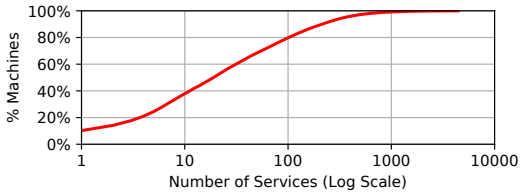


Figure 9: CDF of machines used by services. A small number of services dominate the capacity consumption. Note that the x-axis is in log scale.

Our efficiency effort focuses on these large services, and we find that host-level customization is important for maximizing their performance. For example, customizations help our large web tier achieve 11% higher throughput. However, some custom settings may be beneficial for one service but detrimental to another. As an example, a combination of explicit 2MB and 1GB hugepages improves the web tier's throughput by 4%; however, most services are incapable of utilizing explicit hugepages and enabling this setting globally would lead to unusable memory.

We resolved the conflict between host-level customization and sharing machines in a common pool via *host profiles*, a framework to control host-level customizations on entitlements. An entitlement is associated with one host profile; all machines in the entitlement share the same host profile. When a machine is reassigned from one entitlement to another, *Sidekick* automatically applies the target entitlement's host profile. By fully automating the process of machine allocation and host customization in our shared infrastructure, we can perform fleet-wide optimizations (e.g., swapping machines across entitlements to eliminate hotspots in network or power) without sacrificing workload performance. Supported host profile settings include kernel versions, sysctls (e.g., hugepages and kernel scheduler settings), cgroupv2 (e.g., CPU controller), storage (e.g., XFS or brtfs), NIC settings, CPU Turbo Boost, and hardware prefetch.

2.7 Application-Level Schedulers

As shown at the top of Figure 4, multiple application-level schedulers are built atop Twine to better support vertical workloads such as stateful [16], batch [21], machine learning [13], stream processing [28], and video processing [18]. Twine provides containers as resources for these application-level schedulers to manage and delegates task lifecycle management to them through TaskControl.

Shard Manager (SM) [16] is an example of an application-level scheduler. It is widely used at Facebook to build sharded services like the one in Figure 2. It has two major components: the SM client library and the SM scheduler. The library is linked into a sharded service and provides two APIs for the service to implement: `add_shard()` and `drop_shard()`. The SM scheduler decides the shards each Twine task will host and calls the service's `add_shard()` implementation to prepare the task to serve requests for those shards. To balance load, SM may migrate a shard from task T_1 to task T_2 by informing T_1 to `drop_shard()` and T_2 to `add_shard()`.

The SM scheduler integrates with Twine through TaskControl and can handle the complex situations depicted in Figure 2. In another example, Twine gives SM advance notice about an upcoming maintenance on a machine. If the maintenance duration is short and the shards hosted by the machine have replicas elsewhere, SM may do nothing; otherwise, SM may migrate the impacted shards out of the machine.

2.8 Small Machines and Autoscaling

To achieve higher performance per watt, our server fleet uses millions of small machines [32], each with one 18-core CPU and 64GB RAM. We have worked with Intel to define low-power processors optimized for our environment, e.g., removing unneeded NUMA components. Four small machines are tightly packed into one sled, sharing one multi-host NIC. They are replacing our big machines, each with dual CPUs, 256GB RAM, and a dedicated NIC. Under the same rack-level power budget, a rack holds either 92 small machines or 30 big machines. A small-machine rack delivers 57% higher total compute capacity measured in RRU. Averaged across all our services, using small machines led to 18% savings in power and 17% savings in total cost of ownership (§5.4).

We are consolidating all our compute services onto small machines, as opposed to offering a variety of high-memory or high-CPU machine types. This unification simplifies downstream supply chain and fleet management. It also improves machine fungibility across services, as we can easily reuse a machine across all compute services. Our consolidation journey has been challenging (§7.4), as some services initially did not fit the limited 64GB in our small machines. To address this, we used several common software architectural changes:

- Shard a service so that each instance consumes less memory. Our Shard Manager platform (§2.7) helps developers easily build sharded services running on Twine.
- Exploit data locality to move in-memory data to an external database and use the smaller memory as a cache.
- Exploit data locality to provide tiered memory on top of 64GB RAM and TBs of local flash. For example, when migrating TAO [7], our social graph cache, from big machines to small machines, CacheLib [5] transparently provided tiered memory to improve cache hit ratio and reduce load on the external database by ≈30%.

Our largest services fully utilize small machines without stacking. We rely on Autoscaling to free up underutilized machines. Active Last Minute (ALM) is the number of people who use our online products within a one-minute interval. The load of many services correlates with ALM. *Service Resource Manager* (SRM) uses historical data and realtime measurements to continuously adjust task count for ALM-tracking services and frees up underutilized machines in their entirety for other workloads to use. This work has allowed us to successfully build a large-scale shared infrastructure that consists primarily of small machines.

3 Scaling to One Million Machines

We designed Twine to manage all machines that can fit in a region's 150MW power budget. Although none of our regions host one million machines yet, we are close and anticipate reaching that scale in the near future. Two principles help Twine scale to one million machines and beyond: 1) sharding as opposed to federation, and 2) separation of concerns.

3.1 Scale Out via Sharding

To scale out, we shard Twine schedulers by entitlements, as depicted in Figure 10. We assign newly-created entitlements to shards with the least load. Entitlements can change size and can migrate across shards. If a shard becomes overloaded,

Twine can transparently move an entitlement in the shard to another shard without restarting tasks in the entitlement. Twine can also migrate an individual job from one entitlement to another. To do this, Twine performs a rolling update of the job until all tasks restart on machines belonging to the new entitlement. We automate the execution of these migrations, but humans still decide when and what to migrate. Since migrations happen rarely, we do not automate these further.

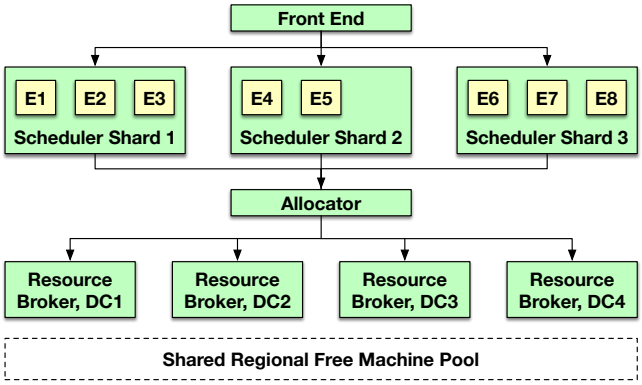


Figure 10: Sharding a scheduler by entitlements. Each scheduler shard manages a different subset of entitlements for the region. `Scheduler Shard 1` manages entitlements `E1`, `E2`, and `E3`. The front end maintains an entitlement-to-shard map and forwards requests to the responsible shards. Each data center has a Resource Broker (RB) managing the machines in that data center. Conceptually, all RBs in a region jointly maintain a free machine pool shared by all entitlements in the region. We also shard the allocator by entitlements and there is a 1:1 mapping between a scheduler shard and an allocator shard. We do not show allocator sharding in the figure as it currently manages a small fraction of our fleet and is still in the process of broader production deployment.

With sharding, the scheduler can easily scale to one million machines. Each data point in Figure 11 plots the P99 CPU utilization of a scheduler shard. The largest shard manages ≈170K machines, using up to 40 cores and 80GB memory. We are moving towards smaller shards to reduce the impact of a shard failure. Assuming each shard manages 50K machines in the future, a single Twine deployment can manage 1M machines with 20 shards. We believe Twine can easily scale beyond 1M machines by adding more shards.

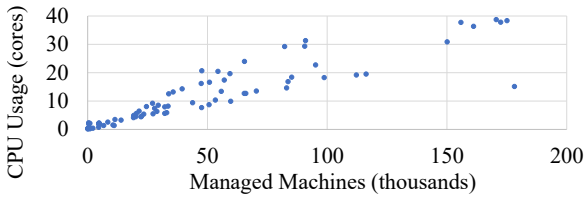


Figure 11: P99 CPU usage of production scheduler shards over one week.

The simplicity of scheduler sharding comes with a theoretical limitation: a single job must fit in a single scheduler shard. This is not a practical limitation. Currently, the largest scheduler shard manages ≈170K machines; the largest entitlement uses ≈60K machines; and the largest job has ≈15K tasks.

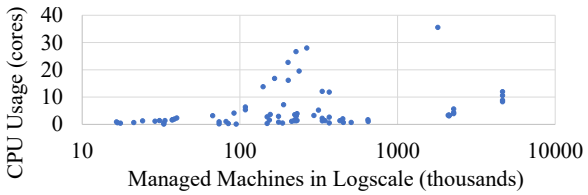


Figure 12: P99 CPU usage of production allocators over one week.

Each data point in Figure 12 plots the P99 CPU utilization of a production allocator. At its peak, a large allocator performs ≈1,000 job allocations per second, with an average job size of 36 tasks. We run a few deployments of the scheduler and allocator at the global level to manage machines and jobs across multiple regions (§7.3). Our largest global allocator currently manages more than one million machines across regions. The allocator is scalable because it has a high cache hit ratio (§2.3), does not handle allocations for short-lived batch jobs (§3.2), and does not perform time-consuming optimizations (§3.2).

3.2 Scale Out via Separation of Concerns

We avoid Kubernetes' centralized architecture where all components interact through one central API server and share one persistent store. These centralized components become bottlenecks and limit Kubernetes' scalability to 5K machines. We shard all Twine components and scale them out independently. Sharded components include the front end, scheduler, allocator, Resource Broker, Health Check Service, and Sidekick. Further, each stateful Twine component (front end, scheduler, allocator, and RB) has its own separate persistent store for metadata. Like Kubernetes [23] and unlike Borg [39], we use external persistent stores for components, as opposed to building the stores directly into components. This allows us to independently shard and scale out persistent stores as needed.

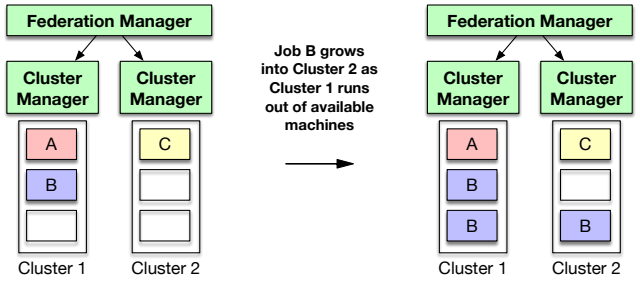
Separation of allocation and optimization responsibilities helps the allocator scale. The allocator makes quick decisions when starting tasks, whereas ReBalancer asynchronously runs a constraint solver to perform time-consuming global optimizations such as balancing CPU, network, and power.

Separation of responsibilities between Twine and application-level schedulers helps Twine scale further. Application-level schedulers handle many fine-grained resource allocation and lifecycle operations without involving Twine. For example, the Twine scheduler and allocator do not directly manage batch jobs, whose lifetime might last just a few seconds and cause high scheduling loads. The application-level batch scheduler acquires resources from Twine in the form of Twine tasks. It reuses these tasks over a long period of time to host different batch jobs, avoiding frequent host profile changes. The batch scheduler can create nested containers inside the tasks, similar to that in Mesos [17].

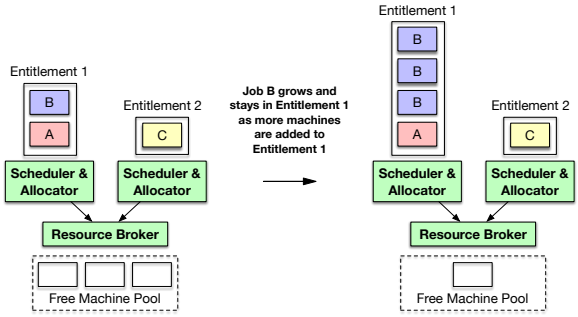
3.3 Comparison of Sharding and Federation

We acknowledge that Twine's scale of managing millions of machines is not unique, as Borg [39] and several public clouds likely manage infrastructure of that scale as well; however, we believe that Twine's approach is unique. Other cluster management systems scale out by deploying one isolated control plane per cluster and operate many siloed clusters. They preallocate machines to a cluster; once a job starts in a cluster, it stays with the cluster. This lack of mobility results in stranded capacity when some clusters are overloaded while others are idle. It also causes operational burden during cluster-wide maintenance such as hardware refresh, as shown in Figure 1.

To avoid stranded capacity, we can introduce mobility by moving either jobs or machines. To that end, the federation approach (e.g., Kubernetes Federation [25]) allows a job to be split across multiple static clusters, whereas Twine dynamically moves machines in and out of entitlements. Figure 13 compares these two approaches.



(a) Federation approach. This approach uses a Cluster Manager per cluster and introduces an additional Federation Manager layer. Each cluster has a set of statically configured machines. As job `B` in `Cluster 1` keeps growing, it overflows into `Cluster 2`.



(b) Twine's sharding approach. As job `B` grows, Twine adds more machines to `Entitlement 1`, and job `B` stays with the same entitlement and scheduler shard.

Figure 13: The two figures above contrast how federation and sharding support a job growing over time without stranding capacity in isolated clusters.

The federation approach can support complex multi-region, hybrid-cloud, or multi-cloud deployments, but it adds complexity as a scale-out solution. In order to provide a seamless user experience, the Federation Manager in Figure 13a has to perform complex coordination for a job whose metadata and management operations are split among multiple distributed Cluster Managers. In contrast, Twine is simpler for scaling out because a job is exclusively managed by one scheduler shard, and Resource Broker provides a simple interface to manage the shared regional pool of machines.

4 Availability and Reliability

Compared with the traditional approach of deploying one control plane per cluster, Twine's regional control plane incurs additional risks: 1) a control plane failure may impact all jobs in a region as opposed to just a cluster, and 2) network partitions may result in a regional Twine scheduler unable to manage an isolated DC.

Design principles. We observe several design principles to mitigate the risks listed above.

- **All components are sharded:** Each shard manages a small fraction of machines and jobs in a region, limiting the impact of a shard failure. Assuming Twine uses 20 scheduler shards to manage a 150MW region, each scheduler shard manages 7.5MW worth of machines, which is no bigger than a traditional cluster.
- **All components are replicated:** Consider schedulers for example: replicas of a scheduler shard sit in different DCs and elect a leader to process requests. If the leader fails or its network is partitioned from other DCs, a follower in another DC becomes the new leader.
- **Tasks keep running:** Even if all Twine components fail, existing tasks continue to run. New jobs cannot be created and existing tasks cannot be updated until Twine recovers. If a DC is partitioned from the scheduler, existing tasks in the DC continue to run.
- **Rate-limit destructive operations:** It is possible that a bug or fault might cause Twine to perform a large number of destructive operations quickly, e.g., shuffling tasks across machines at a fast pace. We protect against this failure by ensuring all components have fail-safe mechanisms to rate-limit destructive operations.
- **Network redundancy:** Fabric Aggregator connects our data centers in a region and can "*suffer many simultaneous failures without compromising the overall performance of the network* [14]." We did not experience within-region network partitioning as a major challenge.

Operational principles. In addition to the design principles listed above, we observe several operational principles.

- **Twine manages itself:** To avoid developing yet another cluster management tool to manage Twine installations, all Twine components, except for the agent, run as Twine jobs. We developed automation to bootstrap the Twine ecosystem starting from scratch. The Twine agent has no dependencies on other Twine components and our bootstrapping mechanism directly sends commands to agents to start other Twine components as Twine tasks.

- **Twine manages its dependencies:** As we built confidence in Twine's bootstrapping automation, we ran all systems that Twine depends on as normal Twine jobs, including ZooKeeper, Delos [4], Configerator [35], and a few other systems for storage, security, and continuous delivery. Twine managing itself and its dependencies improves reliability by eliminating the risk associated with maintaining specialized cluster management tools [8].
- **Gradual but frequent software release:** A new release progresses gradually across regions and shards so that a bug does not hit the entire fleet instantaneously. All components are released weekly or more frequently to lower the risk associated with large changesets.
- **Recurring large-scale failure test [38]:** This happens regularly in production to verify Twine's reliability.

These principles help us run Twine reliably. We share one anecdote where rate-limiting mitigated the risk caused by the complex interplay of four concurrent events: 1) shifting traffic from region X to region Y, 2) performing a load test in region Y, 3) adding new server racks to region Y before removing old racks, and 4) software upgrade for the web tier. The first three events led to increased power consumption in region Y and power capping on many machines. The scheduler rate-limited the number of tasks moving away from power-capped machines. This rate-limiting halted the web tier's software upgrade and protected against further loss of capacity. In this incident, rate-limiting provided a safety net before we debugged the incident.

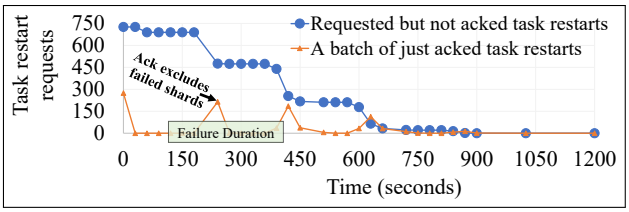
5 Evaluation

Our evaluation answers the following questions:

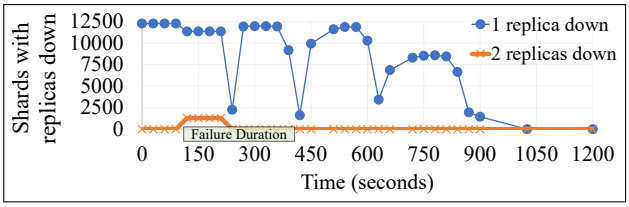
1. How does TaskControl deal with complex scenarios that impact an application's availability?
2. How effective is autoscaling for production use?
3. How effective are host profiles in improving performance? What is the overhead of switching host profiles?
4. How cost effective are small machines in replacing big machines?

5.1 TaskControl

Figure 14 demonstrates how TaskControl handles the complex situation of a software release and machine failures happening concurrently. This experiment uses a caching service managed by Shard Manager (§2.7). The cache's data are partitioned into 15,000 shards, and each shard runs three replicas. The 45,000 shard replicas are hosted by 1,000 Twine tasks. Shard Manager's TaskController helps minimize the risk of a shard losing more than one replica, i.e., driving Figure 14b's `2 replicas down` curve towards zero.



(a) Acknowledged and pending task-restart requests.



(b) Shards with some replicas down.

Figure 14: TaskControl helps a stateful service uphold its availability in the event of a concurrent software release and hardware failures.

Let `T`x denote the moment of x seconds into the experiment. At `T0`, the user initiates a rolling update of the service. In Figure 14a, at `T0`, the TaskController allows 274 tasks to update concurrently (the bottom curve). It does not allow any of the other 726 tasks to update (the top curve) because that would cause some shards to lose their second replicas. In Figure 14b, at `T0`, 12,264 shards lose one replica (the top curve) because they are hosted by the 274 tasks undergoing update. No shard loses its second replica (the bottom curve) because of the TaskController's precise shard availability calculation.

During the `Failure Duration` in the figures (between `T120` and `T415`), we inject the failure of one MSB that kills 50 tasks causing 1,292 shards to lose their second replicas, because those shards are also hosted by the 274 tasks undergoing update. The spike in Figure 14b's bottom curve reflects the impact on the 1,292 shards.

By `T240`, the 274 tasks are updated and become healthy. As a result, even if the 50 tasks in the failed MSB are still down, shards with `2 replicas down` drop to zero (the bottom curve in Figure 14b). At `T240`, the TaskController carefully selects the second batch of 214 tasks to update, ensuring no overlap between the shards hosted by the 214 tasks and the shards hosted by the 50 tasks in the failed MSB (see `Ack excludes failed shards` in Figure 14a). This careful task selection keeps Figure 14b's `2 replicas down` curve at zero throughout the rest of the experiment.

5.2 Autoscaling

Currently, we autoscale ≈800 services. Figure 15 shows the efficacy of autoscaling on our web tier, which is our largest service. Autoscaling frees up to 25% of the web tier's machines during off-peak hours. The bottom curve represents the web tier's CPU utilization. The middle curve represents the web tier's real job size, i.e., the number tasks in the job.

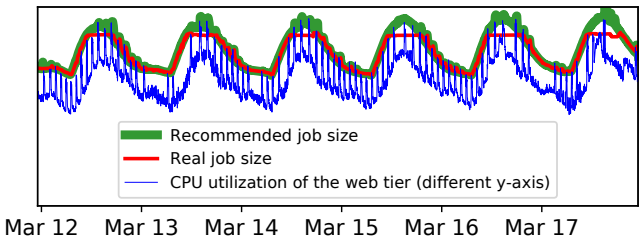


Figure 15: Autoscaling the web tier. The CPU spikes are caused by the continuous-delivery process restarting tasks.

The top curve represents autoscaling's recommendation for the job's ideal size. The CPU utilization closely follows the recommended job size, demonstrating the prediction's accuracy. Usually, the real job size also closely follows the recommended job size, but we intentionally choose a week when they diverged during peak hours.

During the week of March 12, 2020, our online products experienced a drastic traffic growth [19] related to COVID-19, causing a temporary capacity shortage. As a result, the real job size could not grow to follow the recommended job size during peak hours. The web tier's TaskController adapted to this unexpected situation without any manual intervention. During peak hours, it advanced the continuous-delivery software releases more slowly, bringing down fewer tasks concurrently to limit temporary capacity losses. During non-peak hours, it advanced software releases at a normal pace.

5.3 Host Profiles

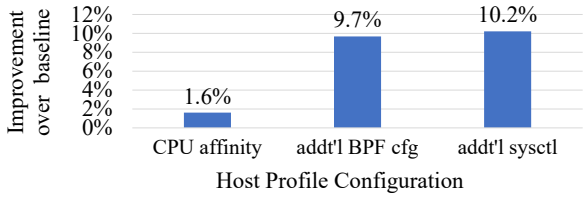


Figure 16: Host profiles improve the throughput of memcache.

Host profile's impact on application performance. We use memcache as an example to demonstrate how host profiles help improve application performance. We deploy a highly optimized version of memcache [30] on tens of thousands of machines. Figure 16 compares three host profiles versus the default settings. The baseline achieves 930K lookups per second on an 18-core/36-hyperthread machine. This extremely high throughput drives the need for host customization.

The `CPU affinity` host profile improves the throughput by dedicating 12 hyperthreads to handling NIC IRQs, one hyperthread to memcache's busy-loop thread, and 23 hyperthreads to memcache's worker threads. This separation avoids unnecessary interrupts and context switches. `Addl't BPF cfg` further reduces the overhead of certain BPF programs by lowering the packet sampling rate and disabling certain packet marking. `Addl't sysctl` further tunes 17 CPU scheduling and network settings, where improvements in reliability are more important than the mild performance gains. For example, based on lessons from past incidents, we tuned `net.ipv4.tcp_mem` to alleviate TCP's memory pressure under high loads in order to prevent cascading failures.

Overhead of switching host profiles. Figure 17 shows the host profile switching time. We discuss both ends of the performance spectrum. The P90 for enabling `CPU Turbo` takes 3.0 seconds. The P90 for enabling `HugePages` takes 244 seconds, as memory fragmentation sometimes causes the Linux kernel to fail to allocate hugepages and a machine reboot may be needed to finish the operation. To alleviate the problem, we recently developed a kernel improvement [33] that achieves above 95% success rate for hugepage allocation; we are still in the process of deploying it to production.

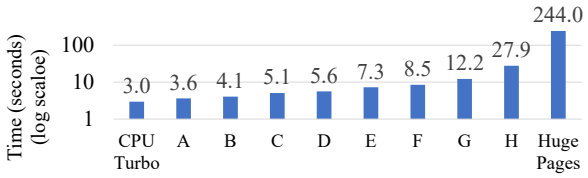


Figure 17: P90 host profile switching time for different host profiles.

On average, a machine changes its host profile once every two days; hence the overall overhead is negligible. Figure 18 depicts how autoscaling impacts host profile changes.

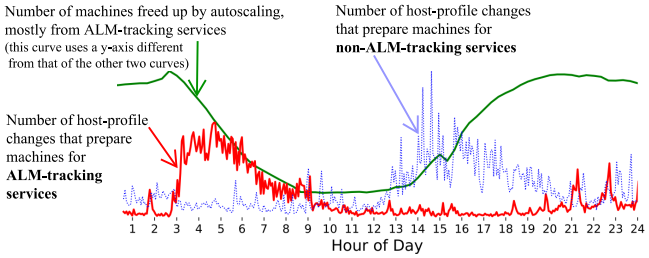


Figure 18: Autoscaling is the biggest driver for host profile changes. The load of an active last minute (ALM) tracking service is proportional to the number of people using our online products. In response to our products' changing load, Twine moves machines to entitlements running ALM-tracking services during hour 3 to 8 and to entitlements running non-ALM-tracking services during hour 13 to 20, respectively.

5.4 Power-efficient Small Machines

The total cost of ownership (TCO) of a machine includes the hardware cost, power consumption, and operating expense. We compare the TCO of small machines vs. big machines using the following metrics:

- *B*: The TCO of a big machine (dual CPUs and 256GB RAM) is *B* times that of a small machine (one CPU and 64GB RAM).

- S: A service needs S number of small machines to replace a big machine and achieve the same performance.
- $\frac{S}{B}$: Relative TCO (RTCO) of a service running on small machines vs. on big machines.

Figure 19 shows the RTCO of 22 fleet-wide representative services. One service has worse than 100% RTCO, seven use the maximum prescribed 100% RTCO, and a majority of services are able to achieve a better RTCO.

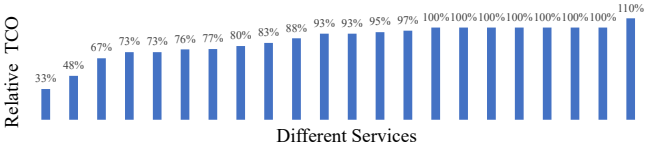


Figure 19: The relative total cost of ownership of services running on small machines vs. on big machines. Smaller numbers mean bigger savings.

The first service in Figure 19 achieves a low 33% RTCO by adopting Shard Manager (§2.7). The service is sharded; its biggest shard has 20x higher load than its smallest shard and the load varies. The service's previous static-sharding solution did not work well, whereas Shard Manager is able to balance the load via shard migration. After switching to small machines, the service better utilizes the overall higher CPU count of small machines under the same TCO.

The second service achieves a 48% RTCO by moving from an in-memory data store to an external flash-based database. Its 48% RTCO includes the cost of the database, which is only a small part of the total TCO.

The service with 76% RTCO is TAO [7], our social graph cache. CacheLib [5] provides transparent tiered memory on top of 64GB RAM and TBs of local flash to replace 256GB RAM (§2.8). Its 76% RTCO includes the cost of flash.

One outlier service has 110% RTCO, meaning it costs 10% more to run on small machines. The memory is used to store certain data indices and ML models that rank the indices. We are improving the service to target 90% RTCO, e.g., by leveraging CacheLib [5] to provide tiered memory.

Across all services in our fleet beyond the examples in Figure 19, we achieved an average 83% RTCO, i.e., 17% fleet-wide TCO savings. This also includes 18% power savings. Overall, we have been successful at using small machines.

6 Experience with Shared Infrastructure

As described in §1, Twine has allowed us to grow *twshared*, our shared compute pool, from ≈15% in 2019 to ≈56% in 2020. We share our experience with growing twshared.

6.1 Economies of Scale in twshared

Shared infrastructure provides economies of scale by reducing hardware, development, and operational costs. Examples:

- **Capacity buffer consolidation.** As services migrated into twshared, we consolidated siloed buffers for software releases, maintenance, fault tolerance, and growth into centralized buffers, improving utilization by ≈3%.
- **Turbo Boost.** We aggressively enabled Turbo on processor cores and relied on ReBalancer to mitigate power hotspots, improving utilization by ≈2% in 2020.
- **Autoscaling.** Autoscaling freed up over-provisioned capacity, reclaiming ≈2% of capacity in 2020.

As shown in Figure 20, as of October 2020, twshared's average memory and CPU utilization are ≈40% and ≈30%, respectively. For comparison, the figure also shows utilization for *private pools*, our legacy pools of customized machines dedicated to individual workloads. We plan to improve utilization through multiple approaches, such as the one described below. Our fleet is dominated by user-facing services that provision capacity for peak load. Autoscaling frees some of this over-provisioned capacity during off-peak hours and provides it as opportunistic capacity for other workloads to use. Unfortunately, we do not yet provide service-level objectives (SLOs) on the availability of opportunistic capacity, which is limiting adoption and usage of all available capacity. As we establish SLOs for opportunistic capacity, improve stacking, and consolidate capacity buffers, we expect twshared's utilization to increase.

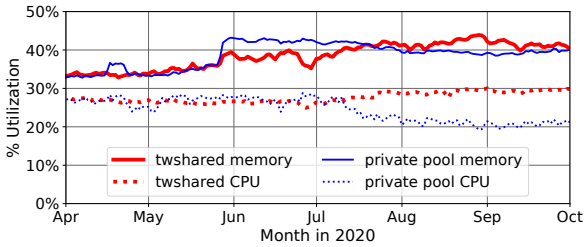


Figure 20: Daily average CPU and memory utilization of twshared and private pools circa October 2020.

6.2 Path to Shared Infrastructure

We had broad conversations with colleagues in industry and learned that while partial consolidation of workloads is common, no large company has achieved near 100% shared infrastructure consolidation. Further, we learned that cultural challenges are as significant as technical challenges. Below, we describe our strategy and major milestones towards migrating *all* non-storage workloads into twshared.

Make Twine capable of supporting a large shared pool. Scalability, entitlements, host profiles, and TaskControl are Twine's important features that enabled workload consolidation. The flexibility offered by host profiles and TaskControl ensures that twshared can support both 1) the general needs of thousands of services, and 2) the specialized needs of a smaller set of services that consume the majority of capacity.

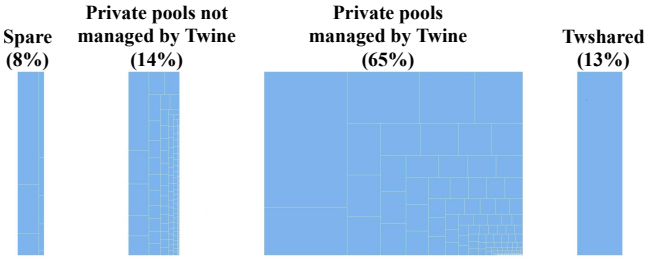


Figure 21: Breakdown of machines in our fleet as of **August 2018**. Each small rectangle inside a category represents a private pool, and its size is proportional to the number of machines in the private pool. There were hundreds of private pools, many of which were small in size. The percentages at the top reflect the number of machines in each category relative to all machines globally. From August 2018 to **October 2020**, the breakdown evolved from [8%, 14%, 65%, 13%] to [13%, 5%, 26%, 56%], where the numbers match the left-to-right categories in the figure.

Publicize the growth and health of twshared. We developed a tool to show the realtime breakdown of our fleet and the growth of twshared. A snapshot is shown in Figure 21. We consolidated the fragmented mechanisms of measuring machine health into the Health Check Service. Continuous improvements have resulted in twshared running healthier than private pools, 99.6% vs. 98.3%.

Set a strong example for others to follow. Early on, we targeted the web tier, our largest private pool. It directly serves external users of our company's products and any outage would be immediately noticeable. We finished migrating the web tier into twshared mid-2019. As the web tier team is highly respected in the company, their testimony motivated others to follow.

Make migration mandatory. After the web tier migration, we gained company-wide support for mandatory migration. Further, we established that all new compute capacity will land only in twshared. This mandate, along with Twine's flexibility of supporting customization through TaskControl and host profiles, has made twshared our ubiquitous compute pool.

6.3 Case Study of twshared Migration

PG_x is a large product group that runs hundreds of diverse services on hundreds of thousands of machines. Their services vary in size from a few machines to tens of thousands, and in complexity from computationally intensive ML training to latency-sensitive ad delivery. Previously, their fleet was fragmented into tens of private pools per region. The first PG_x service migrated into twshared in January 2020; as of September 2020, more than 70% of PG_x machines run in twshared. Given the size and diversity of their services, we expect the migration to finish in late 2021.

PG_x services use hundreds of twshared entitlements; if a service runs in multiple regions, it needs one entitlement per region. Figure 22 shows the size distribution with the biggest entitlement running ≈2K jobs on ≈15K machines.

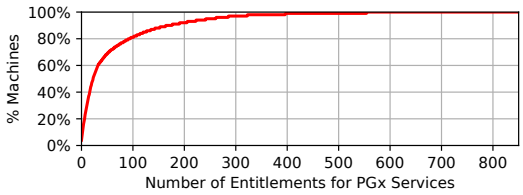


Figure 22: CDF of PG_x entitlement size. The distribution is highly skewed. The largest 54 entitlements account for 70% of PG_x capacity in twshared.

Accommodating workload-specific requirements helps onboard PG_x services onto twshared. For instance, many PG_x services run A/B tests in production, e.g., to evaluate the effectiveness of a new model–these services need to explicitly configure the processor generation for their tasks to prevent performance variations between hardware types from polluting their test results.

The capacity guaranteed by entitlements and private pools account for 55% of PG_x machines. The remaining 45% are from opportunistic sources including capacity buffers, machines freed up by autoscaling, and unused portions of other teams' entitlements. *Optimus* is an application-level scheduler that runs atop Twine to manage opportunistic capacity. When opportunistic capacity is not available, some services gracefully degrade their quality of service.

Jobs with a TaskController consume 36% of PG_x capacity in twshared; in total these jobs use three different TaskControllers, including the one from Shard Manager [16]. About 95% of PG_x capacity is consumed by entitlements that use some combination of these three host profile settings:

1. If a service does frequent flash writes, it prefers the flash drive to expose only a fraction of the flash capacity in order to reduce write amplification and burn rate.
2. If a service can fully utilize a whole machine and does not stack with other services, we disable the cgroup2 CPU controller to eliminate its overhead.
3. Because our data centers are power constrained and CPU Turbo consumes extra power, we enable Turbo only for services that can benefit significantly from Turbo and are running in selected data centers with sufficient power.

Overall, our experience with PG_x indicates that, despite the significant upfront effort needed for migration, even large and varied services are motivated to adopt shared infrastructure that reduces their operational burden. PG_x' success in using opportunistic capacity at a large scale has spurred us to develop SLO guarantees and drive broader adoption (§6.1). Entitlements, TaskControl, and host profiles enable customization in a shared pool and were the features that enabled the migration. On the other hand, PG_x services have grown to hundreds of entitlements within 9 months, motivating us to address entitlement fragmentation (§7.1).

7 Lessons Learned

Evolving Twine and growing twshared has taught us several lessons. We share some highlights and lowlights below.

7.1 Entitlement Fragmentation

We overloaded entitlements with two responsibilities: fleet partitioning and quota management. Entitlements partition millions of machines into smaller units that can be effectively managed by scheduler shards. Twine jobs can only stack within the same entitlement, implying that an entitlement be sized at a few thousand machines, similar to a Borg [39] cell.

On the other hand, leveraging entitlements for quota management results in small entitlements. For example, an important service may wish for an entitlement with 10 tasks rather than a larger entitlement shared with other services to protect against the risk that a rogue service grows unexpectedly and uses up the entitlement quota.

We are in the process of splitting an entitlement's responsibility into two new abstractions: a *materialization* for fleet partitioning and a stackable *reservation* for quota management. A *materialization* functions as a pseudo cluster, has a host profile associated with it, and is always large enough to enable job stacking across thousands of machines.

7.2 Controlled Customization

Our goal is ubiquitous shared infrastructure. A difficult lesson we learned from the first six years of operating twshared was that customization is key to migrating services over. For instance, without host profiles, our web tier and memcache services would not run in twshared as their performance would regress by 11% and 10.2% respectively. TaskControl has provided a path for stateful services such as TAO [7] and MySQL to deprecate their custom cluster management tooling and adopt Twine and shared infrastructure.

We prioritize maintainability when deciding what customization to permit. Currently, we offer 17 host profiles and 16 TaskControllers to support thousands of services. Our recent migration of ≈70% of a large product group's services into twshared (§6.3) leveraged existing host profiles and TaskControllers.

In hindsight, we permitted some customizations that appeared useful initially, but later became barriers for fleet-wide optimizations. For example, a job's tasks are identical by default, but we provided the ability to customize individual tasks, including the executables to run, command line options, environment variables, and restart policies. Developers abused this customization to implement simple sharding so that each task does different work. Autoscaling changes the number of tasks in a job and breaks the job's task customization. As we enable autoscaling for all ALM-tracking services, we are removing task customization and migrating these services to use Shard Manager [16] instead.

7.3 Supporting Global Services

Many developers wish to run a global service without worrying about operational challenges: which regions to deploy to, how much capacity is needed in each region, and how to handle regional failures. We currently operate multiple global Twine deployments that spread a global job's tasks across regions, similar to how a regional Twine deployment spreads a regional job's tasks across data centers in a region. Currently, global jobs account for 8% of all our jobs.

We have learned over time that global Twine deployments did not provide the right abstraction for managing global services. Machines in a region are largely fungible due to the high network bandwidth and low latency within a region, but this is not true for machines distributed across regions. Hence, it is better to explicitly decompose a service's global capacity needs into capacity needs for specific regions, as opposed to global allocators making ad hoc decisions on which regions to get machines from. We are replacing global Twine deployments with a new Federation system built atop regional Twine deployments to provide stronger capacity guarantees and more holistic support for a global-service abstraction.

7.4 Challenges with Small Machines

Our decision to leverage small machines brings with it numerous trade-offs. The effort to rearchitect and reimplement memory-capacity-bound services was higher than we anticipated. On the other hand, we leveraged this opportunity to holistically modernize our legacy services, e.g., moving from static sharding to dynamic sharding for better load balancing. As small machines run contrary to the industry practice of favoring big machines; we need to work closely with hardware vendors to optimize machines for our internal workloads, e.g., removing unneeded NUMA components.

That said, the 18% power efficiency win (§5.4) from small machines has been worth the above trade-offs. We intend to continue using small machines in the coming years, but are also prepared to evolve our hardware strategy as needed. Two factors lead to our decision of adopting small machines: 1) our legacy large services were optimized for utilizing entire machines running in private pools, and 2) our stacking technology needed to mature and improve support for performance isolation [42]. As our services undergo architectural changes to run effectively in twshared, and we improve our stacking technology, we may revisit our hardware strategy.

8 Related Work

Scalability and scheduling performance. Kubernetes [25] and Hydra [9] scale out through federation, whereas Twine scales out through sharding. Figures 13 compares the two approaches. A large body of work [6, 15, 20, 31] focuses on improving batch scheduling throughput and latency. Twine

delegates the handling of short-lived batch jobs to application-level batch schedulers. This separation of concerns helps Twine scale, as discussed in §3.2.

Entitlements. Twine has some similarity to the two-level schedulers (Mesos [17], YARN [37], Apollo [6], and Fuxi [44]), with Twine entitlements as resource offers and Twine scheduler shards as Application Masters (or *frameworks* in Mesos). However, the bottom-level Resource Manager (or *Master* in Mesos) is designed for the scale of a single cluster. In contrast to the single-master two-level architecture, we propose a three-level architecture with sharding so our design scales out: Resource Broker manages machines, Twine scheduler manages containers, and Application-level schedulers manage workloads such as batch and ML.

Kubernetes' cluster autoscaler [24] can respond to workload growth by provisioning VMs in a public cloud and adding them to a node pool. Kubernetes' resizable node pool corresponds to Twine's entitlement, and a public cloud's available resources correspond to Twine's shared free machine pool maintained by Resource Broker. Decoupling Kubernetes and cloud makes the setup flexible, but also misses optimization opportunities compared with Twine's integrated ecosystem. Multiple Kubernetes clusters run independently without coordination, whereas Twine's ReBalancer performs global optimization across entitlements, and an entitlement can be migrated across scheduler shards.

TaskControl. The two-level schedulers (Mesos [17], YARN [37], Apollo [6], and Fuxi [44]) allow their applications to provide custom Application Masters. The interface with Application Masters is for negotiating resource allocation, e.g., "*requesting N containers with X CPU and Y memory*," whereas the TaskControl API is for negotiating lifecycle management, e.g., "*delaying restarting task T*."

Kubernetes [23]'s custom controllers provide a universal extension framework that can be used to implement various custom functions like autoscaling and injecting sidecars for traffic routing. In contrast, TaskControl exclusively focuses on allowing or delaying task lifecycle operations. This narrow interface strikes a balance between standardization and customization (§7.2) and prevents proliferation of customizing all aspects of the Twine control plane. We are unaware of any Kubernetes custom controller that specifically offers extension points to allow or delay task lifecycle operations.

Azure supports update domains and fault domains [3] and the example stateful service in Figure 2 can improve availability by spreading its data shards' replicas across those domains. However, in the event of a machine failure, Azure may still proceed with a rolling update that can lead to unavailable shards because it does not know precisely how the shard replicas are spread across fault domains and update domains.

Host profiles. Paragon [12] schedules a job on machines that are beneficial to the job's performance, but it does not reconfigure a machine.

Some systems statically partition machines in a cluster and preconfigure their hardware and OS settings to suit different workloads. Others dynamically adjust predetermined settings (e.g., Turbo [40]) based on runtime profiling, while disallowing other customizations (e.g., btrfs vs. ext4). We believe that Twine is the first system that 1) allows workloads to provide customized hardware and OS settings to run in a shared machine pool and 2) dynamically reconfigures a machine just-in-time as the workload is scheduled onto the machine. On average, Twine reconfigures a machine once every two days, primarily due to Autoscaling (see Figure 18).

Power-efficient hardware. A large body of work studies power-efficient computing [1, 10, 27]. Our infrastructure is unique in 1) using power-efficient small machines as a universal computing platform, and 2) consolidating towards a single compute machine type (one CPU and 64GB RAM), as opposed to offering a variety of high-memory or high-CPU machine types. Both approaches required our workloads to make software architectural changes that would be challenging in a public cloud with external customer workloads.

Overcommitment and autoscaling. Past work overcommits CPU and memory by colocating batch jobs and online services [11, 22, 39, 43]. Twine does not overcommit CPU or memory by default, although a job owner can explicitly configure their job to do so. On the other hand, we overcommit power by default [41], as power is our most constrained resource. Twine helps mitigate power hotspots by relocating tasks across data centers. Twine's SRM uses historical data to predictably adjust the number of tasks in a job. Borg's Autopilot [34] adjusts the CPU and memory allocated to each task–this is an area of future work for Twine.

9 Conclusion

We identify existing cluster management systems' limitations in supporting large-scale shared infrastructure. We describe our novel solution that allowed us to scale Twine to manage one million machines in a region, move jobs across physical clusters, collaborate with applications to manage their lifecycle, support host customization in a shared pool, use power-efficient small machines to achieve higher performance per watt, and employ autoscaling to improve machine utilization. We share our experience with twshared and our strategy towards ubiquitous shared infrastructure.

Acknowledgments

This paper presents the engineering work of several teams at Facebook that have built Twine and its ecosystem over the past decade. We thank Niket Agarwal, Marius Eriksen, Tianyin Xu, Murray Stokely, Seth Hettich, and the OSDI reviewers for their insightful feedback.

References

[1] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[2] Alexey Andreyev. Introducing data center fabric, the next-generation Facebook data center network, 2014. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[3] Azure update domain and fault domain, 2019. https://docs.microsoft.com/en-us/azure/virtual-machines/availability.

[4] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[5] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[6] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, 2013.

[8] Christopher Bunn. Containerizing ZooKeeper with Twine: Powering container orchestration from within, 2020. Facebook blog post. https://engineering.fb.com/developer-tools/zookeeper-twine/.

[9] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, 2019.

[10] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-Intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[11] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, 2017.

[12] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[13] Jeffrey Dunn. Introducing FBLearner Flow: Facebook's AI backbone, 2016. https://engineering.fb.com/ml-applications/introducing-fblearner-flow-facebook-s-ai-backbone/.

[14] João Ferreira, Naader Hasani, Sreedhevi Sankar, Jimmy Williams, and Nina Schiff. Fabric Aggregator: A flexible solution to our traffic demand, 2014. Facebook blog post. https://engineering.fb.com/data-center-engineering/fabric-aggregator-a-flexible-solution-to-our-traffic-demand/.

[15] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N.M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[16] Gerald Guo and Thawan Kooburat. Scaling services with Shard Manager, 2020. Facebook blog post. https://engineering.fb.com/production-engineering/scaling-services-with-shard-manager/.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott

Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center . In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[18] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito IV, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. SVE: Distributed Video Processing at Facebook Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[19] Mike Isaac and Sheera Frenkel. Facebook Is 'Just Trying to Keep the Lights On' as Traffic Soars in Pandemic. *The New York Times*, 2020. https://www.nytimes.com/2020/03/24/technology/virus-facebook-usage-traffic.html.

[20] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[21] Rui Jian and Hao Lin. Tangram: Distributed Scheduling Framework for Apache Spark at Facebook, 2019. https://databricks.com/session/tangram-distributed-scheduling-framework-for-apache-spark-at-facebook.

[22] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[23] Kubernetes, 2020. https://kubernetes.io/.

[24] Kubernetes cluster autoscaler, 2020. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[25] Kubernetes Federation, 2020. https://github.com/kubernetes/community/tree/master/sig-multicluster.

[26] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sree Kodakara, David Lo, and Partha Ranganathan. Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[27] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V Vasilakos. Cloud Computing: Survey on Energy Efficiency. *Acm computing surveys (csur)*, 47(2):1–36, 2014.

[28] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. Turbine: Facebook's Service Management Platformfor Stream Processing. In *Proceedings of the 36th IEEE International Conference on Data Engineering*, 2020.

[29] Aravind Narayanan, Elisa Shibley, and Mayank Pundir. Fault tolerance through optimal workload placement, 2020. Facebook blog post. https://engineering.fb.com/data-center-engineering/fault-tolerance-through-optimal-workload-placement/.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[31] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[32] Vijay Rao and Edwin Smith. Facebook's new server design delivers on performance without sucking up power, 2016. https://engineering.fb.com/data-center-engineering/facebook-s-new-front-end-server-design-delivers-on-performance-without-sucking-up-power/.

[33] Roman Gushchin. Hugetlb: optionally allocate gigantic hugepages using cma, 2020. https://lkml.org/lkml/2020/3/9/1135.

[34] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at Google. In *Proceedings of the 15th ACM European Conference on Computer Systems*, 2020.

[35] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25nd ACM Symposium on Operating Systems Principles*, 2015.

[36] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of the 15th ACM European Conference on Computer Systems*, 2020.

[37] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013.

[38] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, et al. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[39] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, 2015.

[40] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO Diaries: Application-controlled Frequency Scaling Explained. In *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[41] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's Data Center-Wide Power Management System. *ACM SIGARCH Computer Architecture News*, 44(3), 2016.

[42] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[43] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[44] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale. *Proceedings of the VLDB Endowment*, 7(13):1393–1404, 2014.



FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices

Haoran Qiu[1] Subho S. Banerjee[1] Saurabh Jha[1] Zbigniew T. Kalbarczyk[2]
Ravishankar K. Iyer[1,2]

[1]*Department of Computer Science* [2]*Department of Electrical and Computer Engineering*
University of Illinois at Urbana-Champaign

Abstract

User-facing latency-sensitive web services include numerous distributed, intercommunicating microservices that promise to simplify software development and operation. However, multiplexing of compute resources across microservices is still challenging in production because contention for shared resources can cause latency spikes that violate the service-level objectives (SLOs) of user requests. This paper presents *FIRM*, an intelligent fine-grained resource management framework for predictable sharing of resources across microservices to drive up overall utilization. FIRM leverages online telemetry data and machine-learning methods to adaptively (a) detect/localize microservices that cause SLO violations, (b) identify low-level resources in contention, and (c) take actions to mitigate SLO violations via dynamic reprovisioning. Experiments across four microservice benchmarks demonstrate that FIRM reduces SLO violations by up to 16× while reducing the overall requested CPU limit by up to 62%. Moreover, FIRM improves performance predictability by reducing tail latencies by up to 11×.

1 Introduction

User-facing latency-sensitive web services, like those at Netflix [68], Google [77], and Amazon [89], are increasingly built as microservices that execute on shared/multi-tenant compute resources either as virtual machines (VMs) or as containers (with containers gaining significant popularity of late). These microservices must handle diverse load characteristics while efficiently multiplexing shared resources in order to maintain service-level objectives (SLOs) like end-to-end latency. SLO violations occur when one or more "critical" microservice instances (defined in §2) experience load spikes (due to diurnal or unpredictable workload patterns) or shared-resource contention, both of which lead to longer than expected times to process requests, i.e., latency spikes [4, 11, 22, 30, 35, 44, 53, 69, 98, 99]. Thus, it is critical to efficiently multiplex shared resources among microservices to reduce SLO violations.

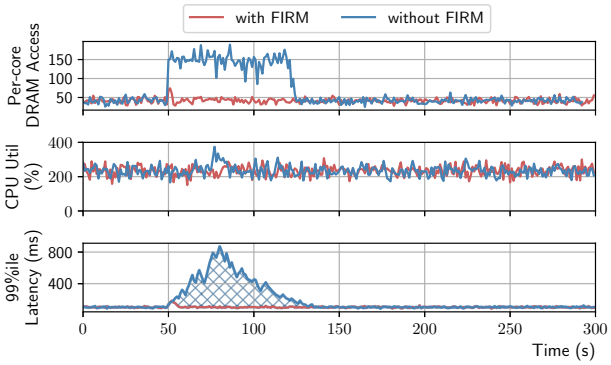


Figure 1: Latency spikes on microservices due to low-level resource contention.

Traditional approaches (e.g., overprovisioning [36, 87], recurrent provisioning [54, 66], and autoscaling [39, 56, 65, 81, 84, 88, 127]) reduce SLO violations by allocating more CPUs and memory to microservice instances by using performance models, handcrafted heuristics (i.e., static policies), or machine-learning algorithms.

Unfortunately, these approaches suffer from two main problems. First, they fail to efficiently multiplex resources, such as caches, memory, I/O channels, and network links, at fine granularity, and thus may not reduce SLO violations. For example, in Fig. 1, the Kubernetes container-orchestration system [20] is unable to reduce the tail latency spikes arising from contention for a shared resource like memory bandwidth, as its autoscaling algorithms were built using heuristics that only monitor CPU utilization, which does not change much during the latency spike. Second, significant human-effort and training are needed to build high-fidelity performance models (and related scheduling heuristics) of large-scale microservice deployments (e.g., queuing systems [27, 39]) that can capture low-level resource contention. Further, frequent microservice updates and migrations can lead to recurring human-expert-driven engineering effort for model reconstruction.

FIRM Framework. This paper addresses the above prob-

lems by presenting *FIRM*, a multilevel machine learning (ML) based resource management (RM) framework to manage shared resources among microservices at finer granularity to reduce resource contention and thus increase performance isolation and resource utilization. As shown in Fig. 1, FIRM performs better than a default Kubernetes autoscaler because FIRM adaptively scales up the microservice (by adding local cores) to increase the aggregate memory bandwidth allocation, thereby effectively maintaining the per-core allocation. FIRM leverages online telemetry data (such as request-tracing data and hardware counters) to capture the system state, and ML models for resource contention estimation and mitigation. Online telemetry data and ML models enable FIRM to adapt to workload changes and alleviate the need for brittle, hand-crafted heuristics. In particular, FIRM uses the following ML models:

- *Support vector machine (SVM) driven detection and localization of SLO violations to individual microservice instances*. FIRM first identifies the "critical paths," and then uses per-critical-path and per-microservice-instance performance variability metrics (e.g., sojourn time [1]) to output a binary decision on whether or not a microservice instance is responsible for SLO violations.
- *Reinforcement learning (RL) driven mitigation of SLO violations that reduces contention on shared resources*. FIRM then uses resource utilization, workload characteristics, and performance metrics to make dynamic reprovisioning decisions, which include (a) increasing or reducing the partition portion or limit for a resource type, (b) scaling up/down, i.e., adding or reducing the amount of resources attached to a container, and (c) scaling out/in, i.e., scaling the number of replicas for services. By continuing to learn mitigation policies through reinforcement, FIRM can optimize for dynamic workload-specific characteristics.

Online Training for FIRM. We developed a *performance anomaly injection framework* that can artificially create resource scarcity situations in order to both train and assess the proposed framework. The injector is capable of injecting resource contention problems at a fine granularity (such as last-level cache and network devices) to trigger SLO violations. To enable rapid (re)training of the proposed system as the underlying systems [67] and workloads [40,42,96,98] change in datacenter environments, FIRM uses *transfer learning*. That is, FIRM leverages transfer learning to train microservice-specific RL agents based on previous RL experience.

Contributions. To the best of our knowledge, this is the first work to provide an SLO violation mitigation framework for microservices by using fine-grained resource management in an application-architecture-agnostic way with multilevel ML models. Our main contributions are:

1. *SVM-based SLO Violation Localization:* We present (in §3.2 and §3.3) an efficient way of localizing the microservice instances responsible for SLO violations by extracting critical paths and detecting anomaly instances in near-real time using telemetry data.
2. *RL-based SLO Violation Mitigation:* We present (in §3.4) an RL-based resource contention mitigation mechanism that (a) addresses the large state space problem and (b) is capable of tuning tailored RL agents for individual microservice instances by using transfer learning.
3. *Online Training & Performance Anomaly Injection:* We propose (in §3.6) a comprehensive performance anomaly injection framework to artificially create resource contention situations, thereby generating the ground-truth data required for training the aforementioned ML models.
4. *Implementation & Evaluation:* We provide an open-source implementation of FIRM for the Kubernetes container-orchestration system [20]. We demonstrate and validate this implementation on four real-world microservice benchmarks [34, 116] (in §4).

Results. FIRM significantly outperforms state-of-the-art RM frameworks like Kubernetes autoscaling [20, 55] and additive increase multiplicative decrease (AIMD) based methods [38, 101].

- It reduces overall SLO violations by up to 16× compared with Kubernetes autoscaling, and 9× compared with the AIMD-based method, while reducing the overall requested CPU by as much as 62%.
- It outperforms the AIMD-based method by up to 9× and Kubernetes autoscaling by up to 30× in terms of the time to mitigate SLO violations.
- It improves overall performance predictability by reducing the average tail latencies up to 11×.
- It successfully localizes SLO violation root-cause microservice instances with 93% accuracy on average.

FIRM mitigates SLO violations without overprovisioning because of two main features. First, it models the dependency between low-level resources and application performance in an RL-based feedback loop to deal with uncertainty and noisy measurements. Second, it takes a two-level approach in which the online critical path analysis and the SVM model filter only those microservices that need to be considered to mitigate SLO violations, thus making the framework application-architecture-agnostic as well as enabling the RL agent to be trained faster.

2 Background & Characterization

The advent of *microservices* has led to the development and deployment of many web services that are composed of "micro," loosely coupled, intercommunicating services, instead of large, monolithic designs. This increased popularity of service-oriented architectures (SOA) of web services has been made possible by the rise of containerization [21, 70, 92, 108] and container-orchestration frameworks [19, 20, 90, 119] that enable modular, low-overhead, low-cost, elastic, and high-efficiency development and production deployment of SOA microservices [8,9,33,34,46,68,77,89,104]. A deployment of

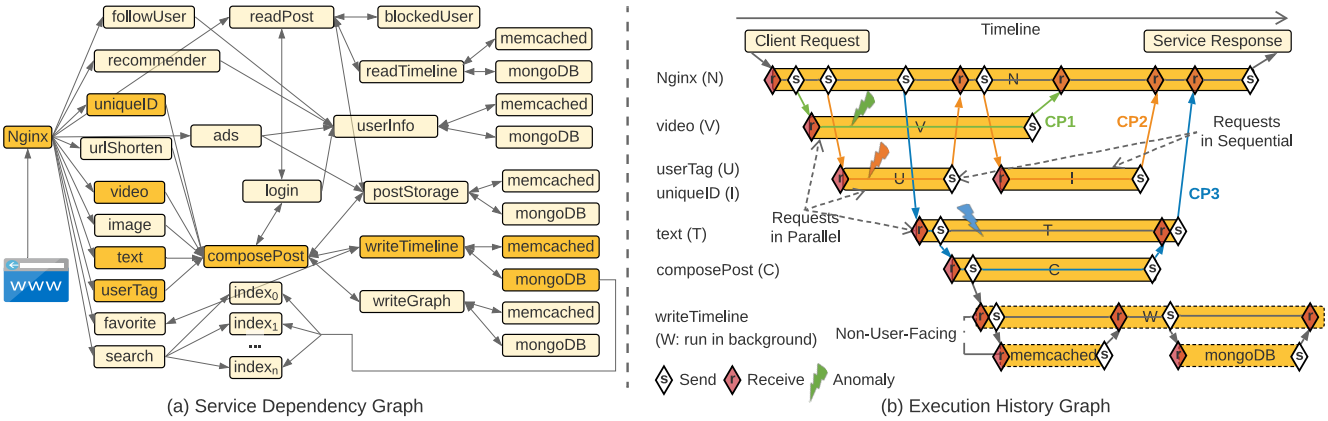
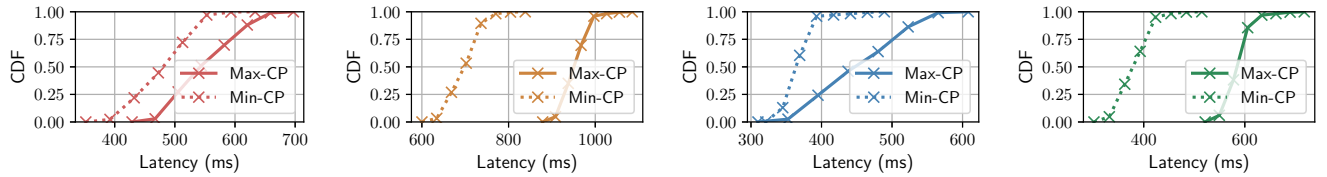


Figure 2: Microservices overview: (a) Service dependency graph of *Social Network* from the DeathStarBench [34] benchmark; (b) Execution history graph of a `post-compose` request in the same microservice.



(a) Social network service. **(b)** Media service. **(c)** Hotel reservation service. **(d)** Train-ticket booking service.

Figure 3: Distributions of end-to-end latencies of different microservices in the DeathStarBench [34] and Train-Ticket [116] benchmarks. Dashed and solid lines correspond to the minimum and maximum critical path latencies on serving a request.

such microservices can be visualized as a *service dependency graph* or an *execution history graph*. The performance of a user request, i.e., its end-to-end latency, is determined by the *critical path* of its execution history graph.

Definition 2.1. A *service dependency graph* captures communication-based dependencies (the edges of the graph) between microservice instances (the vertices of the graph), such as remote procedure calls (RPCs). It tells how requests are flowing among microservices by following parent-child relationship chains. Fig. 2(a) shows the service dependency graph of the *Social Network* microservice benchmark [34]. Each user request traverses a subset of vertices in the graph. For example, in Fig. 2(a), `post-compose` requests traverse only those microservices highlighted in darker yellow.

Definition 2.2. An *execution history graph* is the space-time diagram of the distributed execution of a user request, where a vertex is one of `send_req`, `recv_req`, and `compute`, and edges represent the RPC invocations corresponding to `send_req` and `recv_req`. The graph is constructed using the global view of execution provided by distributed tracing of all involved microservices. For example, Fig. 2(b) shows the execution history graph for the user request in Fig. 2(a).

Definition 2.3. The *critical path* (CP) to a microservice m in the execution history graph of a request is the path of maximal duration that starts with the client request and ends with m [64, 125]. When we mention CP alone without the target microservice m, it means the critical path of the "Service Response" to the client (see Fig. 2(b)), i.e., end-to-end latency.

To understand SLO violation characteristics and study the relationship between runtime performance and the underlying resource contention, we have run extensive performance anomaly injection experiments on widely used microservice benchmarks (i.e. DeathStarBench [34] and Train-Ticket [116]) and collected around 2 TB of raw tracing data (over 4.1×10^7 traces). Our key insights are as follows.

Insight 1: Dynamic Behavior of CPs. In microservices, the latency of the CP limits the overall latency of a user request in a microservice. However, CPs do not remain static over the execution of requests in microservices, but rather change dynamically based on the performance of individual service instances because of underlying shared-resource contention and their sensitivity to this interference. Though other causes may also lead to CP evolution at real-time (e.g., distributed rate limiting [86], and cacheability of requested data [2]), it can still be used as an efficient manifestation of resource interference.

For example, in Fig. 2(b), we show the existence of three different CPs (i.e., CP1–CP3) depending on which microservice (i.e., V, U, T) encounters resource contention. We artificially create resource contention by using *performance*

Table 1: CP changes in Fig. 2(b) under performance anomaly injection. Each case is represented by a <*service*,*CP*> pair. *N*, *V*, *U*, *I*, *T*, and *C* are microservices from Fig. 2.

Case	Average Individual Latency (*ms*)						Total (*ms*)
	N	*V*	*U*	*I*	*T*	*C*	
<*V*,*CP*1>	13	603	166	33	71	68	614 ± 106
<*U*,*CP*2>	14	237	537	39	62	89	580 ± 113
<*T*,*CP*3>	13	243	180	35	414	80	507 ± 75

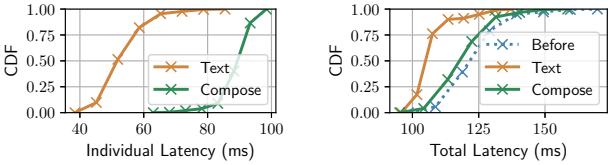


Figure 4: Improvement of end-to-end latency by scaling "highest-variance" and "highest-median" microservices.

anomaly injections.[1] Table 1 lists the changes observed in the latencies of individual microservices, as well as end-to-end latency. We observe as much as 1.2–2× variation in end-to-end latency across the three CPs. Such dynamic behavior exists across all our benchmark microservices. Fig. 3 illustrates the latency distributions of CPs with minimum and maximum latency in each microservice benchmark, where we observe as much as 1.6× difference in median latency and 2.5× difference in 99th percentile tail latency across these CPs.

Recent approaches (e.g., [3,47]) have explored static identification of CPs based on historic data (profiling) and have built heuristics (e.g., application placement, level of parallelism) to enable autoscaling to minimize CP latency. However, our experiment shows that this by itself is not sufficient. The requirement is to *adaptively capture changes in the CPs*, in addition to changing resource allocations to microservice instances on the identified CPs to mitigate tail latency spikes.

Insight 2: Microservices with Larger Latency Are Not Necessarily Root Causes of SLO Violations. It is important to find the microservices responsible for SLO violations to mitigate them. While it is clear that such microservices will always lie on the CP, it is less clear which individual service on the CP is the culprit. A common heuristic is to pick the one with the highest latency. However, we find that that rarely leads to the optimal solution. Consider Fig. 4. The left side shows the CDF of the latencies of two services (i.e., `composePost` and `text`) on the CP of the `post-compose` request in the Social Network benchmark. The `composePost` service has a higher median/mean latency while the `text` service has a higher variance. Now, although the `composePost` service contributes a larger portion of the total latency, it does not benefit from scaling (i.e., getting more resources), as it does not have resource contention. That phenomenon is shown on the right side of Fig. 4, which shows the end-to-end latency for the original configuration (labeled "Before") and after the two microservices were scaled from a single to two containers each (labeled "Text" and "Compose"). Hence, scaling microservices with higher variances provides better performance gain.

[1] *Performance anomaly injections* (§3.6) are used to trigger SLO violations by generating fine-grained resource contention with configurable resource types, intensity, duration, timing, and patterns, which helps with both our characterization (§2) and ML model training (§3.4).

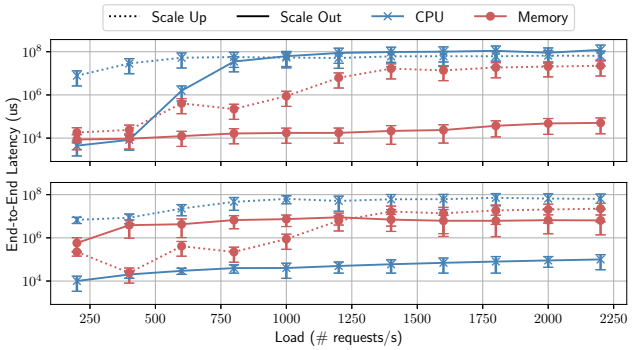


Figure 5: Dynamic behavior of mitigation strategies: *Social Network* (top); *Train-Ticket Booking* (bottom). Error bars show 95% confidence intervals on median latencies.

Insight 3: Mitigation Policies Vary with User Load and Resource in Contention. The only way to mitigate the effects of dynamically changing CPs, which in turn cause dynamically changing latencies and tail behaviors, is to efficiently identify microservice instances on the CP that are resource-starved or contending for resources and then provide them with more of the resources. Two common ways of doing so are (a) to *scale out* by spinning up a new instance of the container on another node of the compute cluster, or (b) to *scale up* by providing more resources to the container via either explicitly partitioning resources (e.g., in the case of memory bandwidth or last-level cache) or granting more resources to an already deployed container of the microservice (e.g., in the case of CPU cores).

As described before, recent approaches [23,38,39,56,65,84,94,101,127]) address the problem by building static policies (e.g., AIMD for controlling resource limits [38, 101], and rule/heuristics-based scaling relying on profiling of historic data about a workload [23, 94]), or modeling performance [39,56]. However, we found in our experiments with the four microservice benchmarks that such static policies are not well-suited for dealing with latency-critical workloads because the optimal policy must incorporate dynamic contextual information. That is, information about the type of user requests, and load (in requests per second), as well as the critical resource bottlenecks (i.e, the resource being contended for), must be jointly analyzed to make optimal decisions. For example, in Fig. 5 (top), we observe that the trade-off between

scale-up and scale-out changes based not only on the user load but also on the resource type. At 500 req/s, scale-up has a better payoff (i.e, lower latency) than scale-out for both memory- and CPU-bound workloads. However, at 1500 req/s, scale-out dominates for CPU, and scale-up dominates for memory. This behavior is also application-dependent because the trade-off curve inflection points change across applications, as illustrated in Fig. 5 (bottom).

3 The FIRM Framework

In this section, we describe the overall architecture of the FIRM framework and its implementation.

1. Based on the insight that resource contention manifests as dynamically evolving CPs, FIRM first detects CP changes and extracts critical microservice instances from them. It does so using the *Tracing Coordinator*, which is marked as (1) in Fig. 6.[2] The tracing coordinator collects tracing and telemetry data from every microservice instance and stores them in a centralized graph database for processing. It is described in §3.1.
2. The *Extractor* detects SLO violations and queries the Tracing Coordinator with collected real-time data (a) to extract CPs (marked as (2) and described in §3.2) and (b) to localize critical microservice instances that are likely causes of SLO violations (marked as (3) and described in §3.3).
3. Using the telemetry data collected in (1) and the critical instances identified in (3), FIRM makes mitigation decisions to scale and reprovision resources for the critical instances (marked as (4)). The policy used to make such decisions is automatically generated using RL. The RL agent jointly analyzes contextual information about resource utilization (i.e., low-level performance counter data collected from the CPU, LLC, memory, I/O, and network), performance metrics (i.e, per-microservice and end-to-end latency distributions), and workload characteristics (i.e., request arrival rate and composition) and makes mitigation decisions. The RL model and setup are described in §3.4.
4. Finally, actions are validated and executed on the underlying Kubernetes cluster through the deployment module (marked as (5) and described in §3.5).
5. In order to train the ML models in the Extractor as well as the RL agent (i.e., to span the exploration-exploitation trade-off space), FIRM includes a performance anomaly injection framework that triggers SLO violations by generating resource contention with configurable intensity and timing. This is marked as (6) and described in §3.6.

3.1 Tracing Coordinator

Distributed tracing is a method used to profile and monitor microservice-based applications to pinpoint causes of poor

[2]Unless otherwise specified, (*) refers to annotations in Fig. 6.

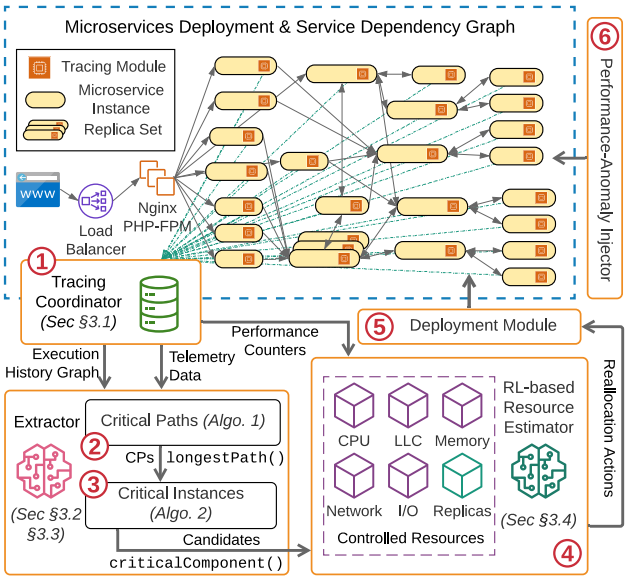


Figure 6: FIRM architecture overview.

performance [111–115]. A *trace* captures the work done by each service along request execution paths, i.e., it follows the execution "route" of a request across microservice instances and records time, local profiling information, and RPC calls (e.g., source and destination services). The execution paths are combined to form the *execution history graph* (see §2). The time spent by a single request in a microservice instance is called its *span*. The span is calculated based on the time when a request arrives at a microservice and when its response is sent back to the caller. Each span is the most basic single unit of work done by a microservice.

The FIRM tracing module's design is heavily inspired by Dapper [95] and its open-source implementations, e.g., Jaeger [112] and Zipkin [115]. Each microservice instance is coupled with an OpenTracing-compliant [75] *tracing agent* that measures spans. As a result, any new OpenTracing-compliant microservice can be integrated naturally into the FIRM tracing architecture. The Tracing Coordinator, i.e., (1), is a stateless, replicable data-processing component that collects the spans of different requests from each tracing agent, combines them, and stores them in a graph database [72] as the execution history graph. The graph database allows us to easily store complex caller-callee relationships among microservices depending on request types, as well as to efficiently query the graph for critical path/component extraction (see §3.2 and §3.3). Distributed clock drift and time shifting are handled using the Jaeger framework. In addition, the Tracing Coordinator collects telemetry data from the systems running the microservices. The data collected in our experiments is listed in Table 2. The distributed tracing and telemetry collection overhead is indiscernible, i.e., we observed a $<0.4\%$ loss in throughput and a $<0.15\%$ loss in latency. FIRM had a

Table 2: Collected telemetry data and sources.

cAdvisor [13] & **Prometheus** [82]
`cpu_usage_seconds_total, memory_usage_bytes,`
`fs_write/read_seconds, fs_usage_bytes,`
`network_transmit/receive_bytes_total, processes`
Linux perf subsystem [79]
`offcore_response.*.llc_hit/miss.local_DRAM,`
`offcore_response.*.llc_hit/miss.remote_DRAM`

maximum CPU overhead of 4.6% for all loads running in our experiments on the four benchmarks [34, 116]. With FIRM, the network in/out traffic without sampling traces increased by 3.4%/10.9% (in bytes); the increase could be less in production environments with larger message sizes [63].

3.2 Critical Path Extractor

The first goal of the FIRM framework is to quickly and accurately identify the CP based on the tracing and telemetry data described in the previous section. Recall from Def. 2.3 in §2 that a CP is the longest path in the request's execution history graph. Hence, changes in the end-to-end latency of an application are often determined by the slowest execution of one or more microservices on its CP.

We identify the CP in an execution history graph by using Alg. 1, which is a weighted longest path algorithm proposed to retrieve CPs in the microservices context. The algorithm needs to take into account the major communication and computation patterns in microservice architectures: (a) *parallel*, (b) *sequential*, and (c) *background* workflows.

- *Parallel workflows* are the most common way of processing requests in microservices. They are characterized by child spans of the same parent span that overlap with each other in the execution history graph, e.g., U, V, and T in Fig. 2(b). Formally, for two child spans i with start time st_i and end time et_i, and j with st_j, et_j of the same parent span p, they are called *parallel* if $(st_j < st_i < et_j) \vee (st_i < st_j < et_i)$.
- *Sequential workflows* are characterized by one or more child spans of a parent span that are processed in a serialized manner, e.g., U and I in Fig. 2(b). For two of p's child-spans i and j to be in a sequential workflow, the time $t_{i\to p} \leq t_{p\to j}$, i.e., i completes and sends its result to p before j does. Such sequential relationships are usually indicative of a *happens-before* relationship. However, it is impossible to ascertain the relationships merely by observing traces from the system. If, across a sufficient number of request executions, there is a violation of that inequality, then the services are not sequential.
- *Background workflows* are those that do not return values to their parent spans, e.g., W in Fig. 2(b). Background workflows are not part of CPs since no other span depends on their execution, but they may be considered responsible for SLO violations when FIRM's Extractor is localizing critical components (see §3.3). That is because background workflows may also contribute to the contention of underlying shared resource.

Algorithm 1 Critical Path Extraction

Require: Microservice execution history graph G
Attributes: *childNodes*, *lastReturnedChild*

procedure LONGESTPATH(G, *currentNode*)
  path ← ∅
  path.add(*currentNode*)
  if *currentNode.childNodes* == None **then**
    Return *path*
  end if
  lrc ← *currentNode.lastReturnedChild*
  path.extend(LONGESTPATH(G, *lrc*))
  for each *cn* in *currentNode.childNodes* **do**
    if *cn*.happensBefore(*lrc*) **then**
      path.extend(LONGESTPATH(G, *cn*))
    end if
  end for
  Return *path*
end procedure

Algorithm 2 Critical Component Extraction

Require: Critical Path CP, Request Latencies T

procedure CRITICALCOMPONENT(G, T)
  $candidates \leftarrow \varnothing$
  $T_{CP} \leftarrow T.getTotalLatency()$ ▷ Vector of CP latencies
  for $i \in CP$ **do**
    $T_i \leftarrow T.getLatency(i)$
    $T_{99} \leftarrow T_i.percentile(99)$
    $T_{50} \leftarrow T_i.percentile(50)$
    $RI \leftarrow PCC(T_i, T_{CP})$ ▷ Relative Importance
    $CI \leftarrow T_{99}/T_{50}$ ▷ Congestion Intensity
    if $SVM.classify(RI, CI) == True$ **then**
      $candidates.append(i)$
    end if
  end for
  Return *candidates*
end procedure

3.3 Critical Component Extractor

In each extracted CP, FIRM then uses an adaptive, data-driven approach to determine critical components (i.e., microservice instances). The overall procedure is shown in Alg. 2. The extraction algorithm first calculates per-CP and per-instance "features," which represent the performance variability and level of request congestion. Variability represents the single largest opportunity to reduce tail latency. The two features are then fed into an incremental SVM classifier to get binary decisions, i.e., on whether that instance should have its resources

re-provisioned or not. The approach is a dynamic selection policy that is in contrast to static policies, as it can classify critical and noncritical components adapting to dynamically changing workload and variation patterns.

In order to extract those microservice instances that are potential candidates for SLO violations, we argue that it is critical to know both the variability of the end-to-end latency (i.e., per-CP variability) and the variability caused by congestion in the service queues of each individual microservice instances (i.e., per-instance variability).

Per-CP Variability: Relative Importance. Relative importance [62, 110, 122] is a metric that quantifies the strength of the relationship between two variables. For each critical path CP, its end-to-end latency is given by $T_{CP} = \sum_{i \in CP} T_i$, where T_i is the latency of microservice i. Our goal is to determine the contribution that the variance of each variable T_i makes toward explaining the total variance of T_{CP}. To do so, we use the Pearson correlation coefficient [12] (also called zero-order correlation), i.e., $PCC(T_i, T_{CP})$, as the measurement, and hence the resulting statistic is known as the variance explained [31]. The sum of $PCC(T_i, T_{CP})$ over all microservice instances along the CP is 1, and the relative importance values of microservices can be ordered by $PCC(T_i, T_{CP})$. The larger the value is, the more variability it contributes to the end-to-end CP variability.

Per-Instance Variability: Congestion Intensity. For each microservice instance in a CP, congestion intensity is defined as the ratio of the 99th percentile latency to the median latency. Here, we chose the 99th percentile instead of the 70th or 80th percentile to target the tail latency behavior. The chosen ratio explains per-instance variability by capturing the congestion level of the request queue so that it can be used to determine whether it is necessary to scale. For example, a higher ratio means that the microservice could handle only a subset of the requests, but the requests at the tail are suffering from congestion issues in the queue. On the other hand, microservices with lower ratios handle most requests normally, so scaling does not help with performance gain. Consequently, microservice instances with higher ratios have a greater opportunity to achieve performance gains in terms of tail latency by taking scale-out or reprovisioning actions.

Implementation. The logic of critical path extraction is incorporated into the construction of spans, i.e., as the algorithm proceeds (Alg. 1), the order of tracing construction is also from the root node to child nodes recursively along paths in the execution history graph. Sequential, parallel, and background workflows are inferred from the parent-child relationships of spans. Then, for each CP, we calculate feature statistics and feed them into an incremental SVM classifier [29,58] implemented using stochastic gradient descent optimization and RBF kernel approximation by `scikit-learn` libraries [91]. Triggered by detected SLO violations, both critical path extraction and critical component extraction are stateless and multithreaded; thus, the workload scales with the size of the microservice application and the cluster. They together constitute FIRM's extractor (i.e., ② and ③). Experiments (§4.2) show that it reports SLO violation candidates with feasible accuracy and achieves completeness with §3.4 by choosing a threshold with a reasonable false-positive rate.

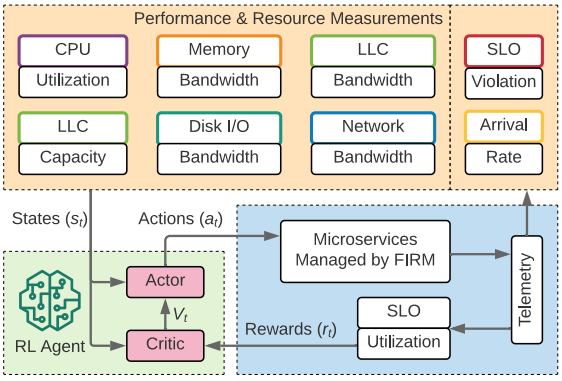


Figure 7: Model-free actor-critic RL framework for estimating resources in a microservice instance.

3.4 SLO Violation Mitigation Using RL

Given the list of critical service instances, FIRM's Resource Estimator, i.e., ④, is designed to analyze resource contention and provide reprovisioning actions for the cluster manager to take. FIRM estimates and controls a fine-grained set of resources, including CPU time, memory bandwidth, LLC capacity, disk I/O bandwidth, and network bandwidth. It makes decisions on scaling each type of resource or the number of containers by using measurements of tracing and telemetry data (see Table 2) collected from the Tracing Coordinator. When jointly analyzed, such data provides information about (a) shared-resource interference, (b) workload rate variation, and (c) request type composition.

FIRM leverages reinforcement learning (RL) to optimize resource management policies for long-term reward in dynamic microservice environments. We next give a brief RL primer before presenting FIRM's RL model.

RL Primer. An RL agent solves a *sequential decision-making problem* (modeled as a Markov decision process) by interacting with an environment. At each discrete time step t, the agent observes a *state of the environment* $s_t \in S$, and performs an *action* $a_t \in A$ based on its *policy* $\pi_\theta(s)$ (parameterized by θ), which maps *state space* S to *action space* A. At the following time step $t+1$, the agent observes an *immediate reward* $r_t \in R$ given by a reward function $r(s_t, a_t)$; the immediate reward represents the loss/gain in transitioning from s_t to s_{t+1} because of action a_t. The tuple (s_t, a_t, r_t, s_{t+1}) is called one *transition*. The agent's goal is to optimize the policy π_θ so as to maximize the expected *cumulative discounted reward* (also called the value function) from the start distribution $J = \mathbb{E}[G_1]$, where the return from a state G_t is defined

to be $\sum_{k=0}^{T}\gamma^k r_{t+k}$. The discount factor $\gamma \in (0,1]$ penalizes the predicted future rewards.

Two main categories of approaches are proposed for policy learning: value-based methods and policy based methods [5]. In value-based methods, the agent learns an estimate of the optimal value function and approaches the optimal policy by maximizing it. In policy-based methods, the agent directly tries to approximate the optimal policy.

Why RL? Existing performance-modeling-based [23, 38, 39, 56, 94, 101, 127] or heuristic-based approaches [6, 7, 37, 65, 84] suffer from model reconstruction and retraining problems because they do not address dynamic system status. Moreover, they require expert knowledge, and it takes significant effort to devise, implement, and validate their understanding of the microservice workloads as well as the underlying infrastructure. RL, on the other hand, is well-suited for learning resource reprovisioning policies, as it provides a tight feedback loop for exploring the action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules). It allows direct learning from actual workload and operating conditions to understand how adjusting low-level resources affects application performance. In particular, FIRM utilizes the deep deterministic policy gradient (DDPG) algorithm [59], which is a *model-free*, *actor-critic* RL framework (shown in Fig. 7). Further, FIRM's RL formulation provides two distinct advantages:

1. Model-free RL does not need the ergodic distribution of states or the environment dynamics (i.e., transitions between states), which are difficult to model precisely. When microservices are updated, the simulations of state transitions used in model-based RL are no longer valid.
2. The Actor-critic framework combines policy-based and value-based methods (i.e., consisting of an actor-net and a critic-net as shown in Fig. 8), and that is suitable for continuous stochastic environments, converges faster, and has lower variance [41].

Learning the Optimal Policy. DDPG's policy learning is an actor-critic approach. Here the "critic" estimates the *value function* (i.e., the expected value of cumulative discounted reward under a given policy), and the "actor" updates the policy in the direction suggested by the critic. The critic's estimation of the expected return allows the actor to update with gradients that have lower variance, thus speeding up the learning process (i.e., achieving convergence). We further assume that the actor and critic are represented as deep neural networks. DDPG also solves the issue of dependency between samples and makes use of hardware optimizations by introducing a *replay buffer*, which is a finite-sized cache $\mathcal{D}$ that stores transitions (s_t, a_t, r_t, s_{t+1}). Parameter updates are based on a mini-batch of size N sampled from the reply buffer. The pseudocode of the training algorithm is shown in Algorithm 3. RL training proceeds in episodes and each episode consists of T time steps. At each time step, both actor and critic neural nets are updated once.

Algorithm 3 DDPG Training

```
Randomly init Q_w(s,a) and π_θ(a|s) with weights w & θ.
Init target network Q' and π' with w' ← w & θ' ← θ
Init replay buffer D ← ∅
for episode = 1, M do
    Initialize a random process N for action exploration
    Receive initial observation state s_1
    for t = 1, T do
        Select and execute action a_t = π_θ(s_t) + N_t
        Observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Sample N transitions (s_i, a_i, r_i, s_{i+1}) from D
        Update critic by minimizing the loss L(w)
        Update actor by sampled policy gradient ∇_θ J
        w' ← γw + (1−γ)w'
        θ' ← γθ + (1−γ)θ'
    end for
end for
```

In the critic, the value function $Q_w(s_t, a_t)$ with parameter w and its corresponding loss function are defined as:

$$Q_w(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma Q_w(s_{t+1}, \pi(s_{t+1}))]$$

$$\mathcal{L}(w) = \frac{1}{N}\sum_i (r_i + \gamma Q'_{w'}(s_{i+1}, \pi'_{\theta'}(s_{i+1})) - Q_w(s_i, a_i))^2.$$

The target networks $Q'_{w'}(s,a)$ and $\pi'_{\theta'}(s)$ are introduced in DDPG to mitigate the problem of instability and divergence when one is directly implementing deep RL agents. In the actor component, DDPG maintains a parametrized actor function $\pi_\theta(s)$, which specifies the current policy by deterministically mapping states to a specific action. The actor is updated as follows:

$$\nabla_\theta J = \frac{1}{N}\sum_i \nabla_a Q_w(s = s_i, a = \pi(s_i)) \nabla_\theta \pi_\theta(s = s_i).$$

Problem Formulation. To estimate resources for a microservice instance, we formulate a sequential decision-making problem which can be solved by the above RL framework. Each microservice instance is deployed in a separate container with a tuple of resource limits $RLT = (RLT_{cpu}, RLT_{mem}, RLT_{llc}, RLT_{io}, RLT_{net})$, since we are considering CPU utilization, memory bandwidth, LLC capacity, disk I/O bandwidth, and network bandwidth as our resource model.[3] This limit for each type of resource is predetermined (usually overprovisioned) before the microservices are deployed in the cluster and later controlled by FIRM.

At each time step t, utilization RU_t for each type of resource is retrieved using performance counters as telemetry data in ①. In addition, FIRM's Extractor also collects current

[3] The resource limit for the CPU utilization of a container is the smaller of $\hat{R}_i$ and the number of threads $\times$ 100.

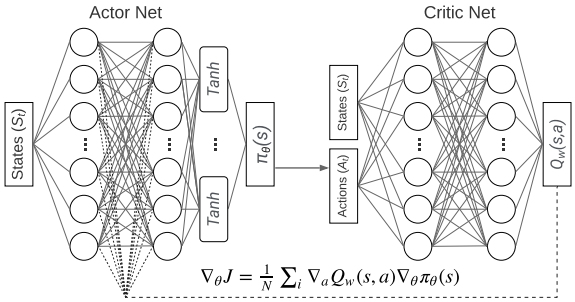


Figure 8: Architecture of actor-critic nets.

latency, request arrival rate, and request type composition (i.e., percentages of each type of request). Based on these measurements, the RL agent calculates the states listed in Table 3 and described below.

- *SLO maintenance ratio* (SM_t) is defined as `SLO_latency/current_latency` if the microservice instance is determined to be the culprit. If no message arrives, it is assumed that there is no SLO violation ($SM_t = 1$).
- *Workload changes* (WC_t) is defined as the ratio of the arrival rates at the current and previous time steps.
- *Request composition* (RC_t) is defined as a unique value encoded from an array of request percentages by using `numpy.ravel_multi_index()` [74].

For each type of resources i, there is a predefined resource upper limit $\hat{R}_i$ and a lower limit $\underset{\sim}{R}_i$ (e.g., the CPU time limit cannot be set to 0). The actions available to the RL-agent is to set $RLT_i \in [\hat{R}_i, \underset{\sim}{R}_i]$. If the amount of resource reaches the total available amount, then a scale-out operation is needed. Similarly, if the resource limit is below the lower bound, a scale-in operation is needed. The CPU resources serve as one exception to the above procedure: it would not improve the performance if the CPU utilization limit were higher than the number of threads created for the service.

The goal of the RL agent is, given a time duration t, to determine an optimal policy π_t that results in as few SLO violations as possible (i.e., $\min_{\pi_t} SM_t$) while keeping the resource utilization/limit as high as possible (i.e., $\max_{\pi_t} RU_t/RLT_t$). Based on both objectives, the reward function is then defined as $r_t = \alpha \cdot SM_t \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_i^{|\mathcal{R}|} RU_i/RLT_i$, where $\mathcal{R}$ is the set of resources.

Transfer Learning. Using a tailored RL agent for every microservice instead of using the shared RL agent should improve resource reprovisioning efficiency, as the model would be more sensitive to application characteristics and features. However, such an approach is hard to justify in practice (i.e., for deployment) because of the time required to train such tailored models for user workloads, which might have significant churn. FIRM addresses the problem of rapid model training by using transfer learning in the domain of RL [14, 105, 106], whereby agents for SLO violation mitigation can be trained for either the general case (i.e., any microservices) or the specialized case (i.e., "transferred" to the behavior of individualized microservices). The pre-trained model used in the specialized case is called the base model or the source model. That approach is possible because prior understanding of a problem structure helps one solve similar problems quickly, with the remaining task being to understand the behavior of updated microservice instances. Related work on base model selection and task similarity can be found in [105, 106], but the base model that FIRM uses for transfer learning is always the RL model learned in the general case because it has been shown in evaluation to be comparable with specialized models. We demonstrate the efficacy of transfer learning in our evaluation described in §4. The RL model that FIRM uses is designed to scale since both the state space and the action space are independent of the size of the application or the cluster. In addition to having the general case RL agent, the FIRM framework also allows for the deployment of specialized per-microservice RL agents.

Table 3: State-action space of the RL agent.

State (s_t)
SLO Maintenance Ratio (SM_t), Workload Changes (WC_t), Request Composition (RC_t), Resource Utilization (RU_t)
Action Space (a_t)
Resource Limits $RLT_i(t), i \in \{\text{CPU, Mem, LLC, IO, Net}\}$

Table 4: RL training parameters.

Parameter	**Value**
# Time Steps × # Minibatch	300×64
Size of Replay Buffer	10^5
Learning Rate	Actor (3×10^{-4}), Critic (3×10^{-3})
Discount Factor	0.9
Soft Update Coefficient	2×10^{-3}
Random Noise	μ (0), σ (0.2)
Exploration Factor	ε (1.0), ε-decay (10^{-6})

Implementation Details. We implemented the DDPG training algorithm and the actor-critic networks using `PyTorch` [83]. The critic net contains two fully connected hidden layers with 40 hidden units, all using ReLU activation function. The first two hidden layers of the actor net are fully connected and both use ReLU as the activation function while the last layer uses Tanh as the activation function. The actor network has 8 inputs and 5 outputs, while the critic network has 23 inputs and 1 output. The actor and critic networks are shown in Fig. 8, and their inputs and outputs are listed in Table 3. We chose that setup because adding more layers and hidden units does not increase performance in our experiments with selected microservice benchmarks; instead, it slows down training speed significantly. Hyperparameters of the RL model are listed in Table 4. We set the time step for training the model to be 1 second, which is sufficient for action execution (see Table 6). The latencies of each RL train-

Table 5: Types of performance anomalies injected to microservices causing SLO violations.

Performance Anomaly Types	Tools/Benchmarks
Workload Variation	`wrk2` [123]
Network Delay	`tc` [107]
CPU Utilization	iBench [24], `stress-ng` [100]
LLC Bandwidth & Capacity	iBench, `pmbw` [80]
Memory Bandwidth	iBench [24], `pmbw` [80]
I/O Bandwidth	Sysbench [102]
Network Bandwidth	`tc` [107], Trickle [117]

ing update and inference step are 73 $\pm$ 10.9 ms and 1.2 $\pm$ 0.4 ms, respectively. The average CPU and memory usage of the Kubernetes pod during the training stage are 210 millicores and 192 Mi, respectively.

3.5 Action Execution

FIRM's Deployment Module, i.e., (5), verifies the actions generated by the RL agent and executes them accordingly. Each action on scaling a specific type of resource is limited by the total amount of the resource available on that physical machine. FIRM assumes that machine resources are unlimited and thus does not have admission control or throttling. If an action leads to oversubscribing of a resource, then it is replaced by a scale-out operation.

- *CPU Actions:* Actions on scaling CPU utilization are executed through modification of `cpu.cfs_period_us` and `cpu.cfs_quota_us` in the `cgroups` CPU subsystem.
- *Memory Actions:* We use Intel MBA [49] and Intel CAT [48] technologies to control the memory bandwidth and LLC capacity of containers, respectively.[4]
- *I/O Actions:* For I/O bandwidth, we use the `blkio` subsystem in `cgroups` to control input/output access to disks.
- *Network Actions:* For network bandwidth, we use the Hierarchical Token Bucket (HTB) [45] queueing discipline in Linux Traffic Control. Egress `qdiscs` can be directly shaped by using HTB. Ingress `qdiscs` are redirected to the virtual device `ifb` interface and then shaped through the application of egress rules.

3.6 Performance Anomaly Injector

We accelerate the training of the machine learning models in FIRM's Extractor and the RL agent through performance anomaly injections. The injection provides the ground truth data for the SVM model, as the injection targets are controlled and known from the campaign files. It also allows the RL agent to quickly span the space of adverse resource contention behavior (i.e., the exploration-exploitation trade-off in RL). That is important, as real-world workloads might not experience all adverse situations within a short training time. We implemented a performance anomaly injector, i.e., (6), in which the injection targets, type of anomaly, injection time, duration, patterns, and intensity are configurable. The injector is designed to be bundled into the microservice containers as a file-system layer; the binaries incorporated into the container can then be triggered remotely during the training process. The injection campaigns (i.e., how the injector is configured and used) for the injector will be discussed in §4. The injector comprises seven types of performance anomalies that can cause SLO violations. They are listed in Table 5 and described below.

Workload Variation. We use an HTTP benchmarking tool `wrk2` as the workload generator. It performs multithreaded, multiconnection HTTP request generation to simulate client-microservice interaction. The request arrival rate and distribution can be adjusted to break the predefined SLOs.

Network Delay. We use Linux traffic control (`tc`) to add simulated delay to network packets. Given the mean and standard deviation of the network delay latency, each network packet is delayed following a normal distribution.

CPU Utilization. We implement the CPU stressor based on iBench and `stree-ng` to exhaust a specified level of CPU utilization on a set of cores by exercising floating point, integer, bit manipulation and control flows.

LLC Bandwidth & Capacity. We use iBench and `pmbw` to inject interference on the Last Level Cache (LLC). For bandwidth, the injector performs streaming accesses in which the size of the accessed data is tuned to the parameters of the LLC. For capacity, it adjusts intensity based on the size and associativity of the LLC to issue random accesses that cover the LLC capacity.

Memory Bandwidth. We use iBench and `pmbw` to generate memory bandwidth contention. It performs serial memory accesses (of configurable intensity) to a small fraction of the address space. Accesses occur in a relatively small fraction of memory in order to decouple the effects of contention in memory bandwidth from contention in memory capacity.

I/O Bandwidth. We use Sysbench to implement the file I/O workload generator. It first creates test files that are larger than the size of system RAM. Then it adjusts the number of threads, read/write ratio, and sleeping/working ratio to meet a specified level of I/O bandwidth. We also use Tricle for limiting the upload/download rate of a specific microservice instance.

Network Bandwidth. We use Linux traffic control (`tc`) to limit egress network bandwidth. For ingress network bandwidth, an intermediate function block (`ifb`) pseudo interface is set up, and inbound traffic is directed through that. In that way, the inbound traffic then becomes schedulable by the egress `qdisc` on the `ifb` interface, so the same rules for egress can be applied directly to ingress.

[4] Our evaluation on IBM Power systems (see §4) did not use these actions because of a lack of hardware support. OS support or software partitioning mechanisms [60, 85] can be applied; we leave that to future work.

4 Evaluation

4.1 Experimental Setup

Benchmark Applications. We evaluated FIRM on a set of end-to-end interactive and responsive real-world microservice benchmarks: (i) DeathStarBench [34], consisting of *Social Network*, *Media Service*, and *Hotel Reservation* microservice applications, and (ii) Train-Ticket [128], consisting of the *Train-Ticket Booking Service*. *Social Network* implements a broadcast-style social network with unidirectional follow relationships whereby users can publish, read, and react to social media posts. *Media Service* provides functionalities such as reviewing, rating, renting, and streaming movies. *Hotel Reservation* is an online hotel reservation site for browsing hotel information and making reservations. *Train-Ticket Booking Service* provides typical train-ticket booking functionalities, such as ticket inquiry, reservation, payment, change, and user notification. These benchmarks contain 36, 38, 15, and 41 unique microservices, respectively; cover all workflow patterns (see §3.2); and use various programming languages including Java, Python, Node.js, Go, C/C++, Scala, PHP, and Ruby. All microservices are deployed in separate Docker containers.

System Setup. We validated our design by implementing a prototype of FIRM that used Kubernetes [20] as the underlying container orchestration framework. We deployed the four microservice benchmarks with FIRM separately on a Kubernetes cluster of 15 two-socket physical nodes without specifying any anti-colocation rules. Each server consists of 56–192 CPU cores and RAM that varies from 500 GB to 1000 GB. Nine of the servers use Intel x86 Xeon E5s and E7s processors, while the remaining ones use IBM ppc64 Power8 and Power9 processors. All machines run Ubuntu 18.04.3 LTS with Linux kernel version 4.15.

Load Generation. We drove the services with various open-loop asynchronous workload generators [123] to represent an active production environment [17, 97, 118]. We uniformly generated workloads for every request type across all microservice benchmarks. The parameters for the workload generators were the same as those for DeathStarBench (which we applied to Train-Ticket as well), and varied from predictable constant, diurnal, distributions such as Poisson, to unpredictable loads with spikes in user demand. The workload generators and the microservice benchmark applications were never co-located (i.e., they executed on different nodes in the cluster). To control the variability in our experiments, we disabled all other user workloads on the cluster.

Injection and Comparison Baselines. We used our performance anomaly injector (see §3.6) to inject various types of performance anomalies into containers uniformly at random with configurable injection timing and intensity. Following the common way to study resource interference, our experiments on SLO violation mitigation with anomalies were designed to be comprehensive by covering the worst-case scenarios, given the random and nondeterministic nature of shared-resource interference in production environments [22, 78]. Unless otherwise specified, (i) the anomaly injection time interval was in an exponential distribution with $\lambda = 0.33s^{-1}$, and (ii) the anomaly type and intensity were selected uniformly at random. We implemented two baseline approaches: (a) the Kubernetes autoscaling mechanism [55] and (b) an AIMD-based method [38, 101] to manage resources for each container. Both approaches are rule-based autoscaling techniques.

4.2 Critical Component Localization

Here, we use the techniques presented in §3.2 and §3.3 to study the effectiveness of FIRM in identifying the microservices that are most likely to cause SLO violations.

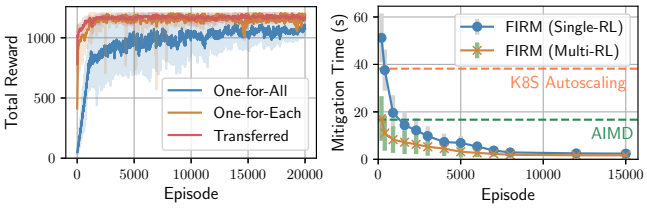
Single anomaly localization. We first evaluated how well FIRM localizes the microservice instances that are responsible for SLO violations under different types of single-anomaly injections. For each type of performance anomaly and each type of request, we gradually increased the intensity of injected resource interference and recorded end-to-end latencies. The intensity parameter was chosen uniformly at random between [start-point, end-point], where the start-point is the intensity that starts to trigger SLO violations, and the end-point is the intensity when either the anomaly injector has consumed all possible resources or over 80% of user requests have been dropped or returned time. Fig. 9(a) shows the receiver operating characteristic (ROC) curve of root cause localization. The ROC curve captures the relationship between the false-positive rate (x-axis) and the true-positive rate (y-axis). The closer to the upper-left corner the curve is, the better the performance. We observe that the localization accuracy of FIRM, when subject to different types of anomalies, does not vary significantly. In particular, FIRM's Extractor module achieved near 100% true-positive rate, when the false-positive rate was between $[0.12, 0.16]$.

Multi-anomaly localization. There is no guarantee that only one resource contention will happen at a time under dynamic datacenter workloads [40, 42, 96, 98] and therefore we also studied the container localization performance under multi-anomaly injections and compared machines with two different processor ISAs (x86 and ppc64). An example of the intensity distributions of all the anomaly types used in this experiment are shown in Fig. 9(c). The experiment was divided into time windows of 10 s, i.e., T_i from Fig. 9(c)). At each time window, we picked the injection intensity of each anomaly type uniformly at random with range [0,1]. Our observations are reported in Fig. 9(b). The average accuracy for localizing critical components in each application ranged from 92% to 94%. The overall average localization accuracy was 93% across four microservice benchmarks. Overall, we observe that the accuracy of the Extractor did not differ between the two sets of processors.



(a) ROC under single-anomaly. **(b)** Average accuracy under multi-anomaly. **(c)** Anomaly injection intensity and timing.

Figure 9: Critical Component Localization Performance: (a) ROC curves for detection accuracy; (b) Variation of localization accuracies across processor architectures; (c) Anomaly-injection intensity, types, and timing.



(a) Total reward. **(b)** SLO mitigation time.

Figure 10: Learning curve showing total reward during training and SLO mitigation performance.

4.3 RL Training & SLO Violation Mitigation

To understand the convergence behavior of FIRM's RL agent, we trained three RL models that were subjected to the same sequence of performance anomaly injections (described in §4.1). The three RL models are: (i) a common RL agent for all microservices (one-for-all), (ii) a tailored RL agent for a particular microservice (one-for-each), and (iii) a transfer-learning-based RL agent. RL training proceeds in episodes (iterations). We set the number of time steps in a training episode to be 300 (see Table 4), but for the initial stages, we terminate the RL exploration early so that the agent could reset and try again from the initial state. We did so because the initial policies of the RL agent are unable to mitigate SLO violations. Continuously injecting performance anomalies causes user requests to drop, and thus only a few request traces were generated to feed the agent. As the training progressed, the agent improved its resource estimation policy and could mitigate SLO violations in less time. At that point (around 1000 episodes), we linearly increased the number of time steps to let the RL agent interact with the environment longer before terminating the RL exploration and entering the next iteration.

We trained the abovementioned three RL models on the Train-Ticket benchmark. We studied the generalization of the RL model by evaluating the end-to-end performance of FIRM on the DeathStarBench benchmarks. Thus, we used DeathStarBench as a validation set in our experiments. Fig. 10(a) shows that as the training proceeded, the agent was getting better at mitigation, and thus the moving average of episode rewards was increasing. The initial steep increase benefits from early termination of episodes and parameter exploration. Transfer-learning-based RL converged even faster (around 2000 iterations[5]) because of parameter sharing. The one-for-all RL required more iterations to converge (around 15000 iterations) and had a slightly lower total reward (6% lower compared with one-for-each RL) during training.

In addition, higher rewards, for which the learning algorithm explicitly optimizes, correlate with improvements in SLO violation mitigation (see Fig. 10(b)). For models trained in every 200 episodes, we saved the checkpoint of parameters in the RL model. Using the parameter, we evaluated the model snapshot by injecting performance anomalies (described in §4.1) continuously for one minute and observed when SLO violations were mitigated. Fig. 10(b) shows that FIRM with either a single-RL agent (one-for-all) or a multi-RL agent (one-for-each) improved with each episode in terms of the SLO violation mitigation time. The starting policy at iteration 0–900 was no better than the Kubernetes autoscaling approach, but after around 2500 iterations, both agents were better than either Kubernetes autoscaling or the AIMD-based method. Upon convergence, FIRM with a single-RL agent achieved a mitigation time of 1.7 s on average, which outperformed the AIMD-based method by up to 9× and Kubernetes autoscaling by up to 30× in terms of the time to mitigate SLO violations.

4.4 End-to-End Performance

Here, we show the end-to-end performance of FIRM and its generalization by further evaluating it on DeathStarBench benchmarks based on the hyperparameter tuned during training with the Train-Ticket benchmark. To understand the 10–30× improvement demonstrated above, we measured the 99th percentile end-to-end latency when the microservices were being managed by the two baseline approaches and by FIRM. Fig. 11(a) shows the cumulative distribution of the end-to-end

[5] 1000 iterations correspond to roughly 30 minutes with each iteration consisting of 300 time steps.

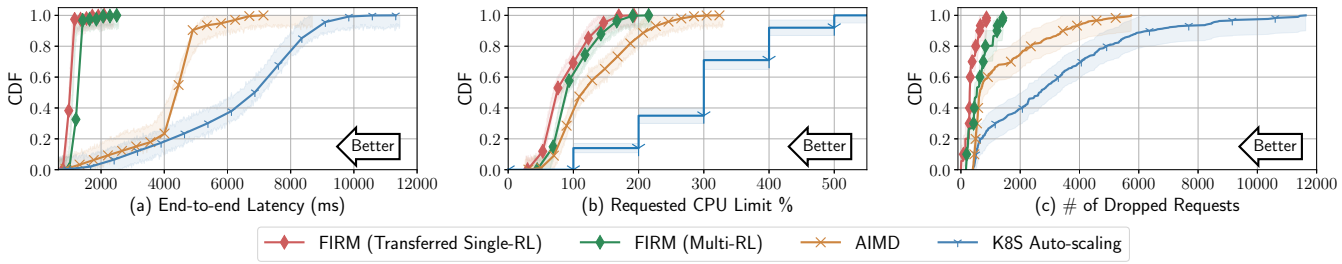


Figure 11: Performance comparisons (CDFs) of end-to-end latency, requested CPU limit, and the number of dropped requests.

latency. We observed that the AIMD-based method, albeit simple, outperforms the Kubernetes autoscaling approach by 1.7× on average and by 1.6× in the worst case. In contrast, FIRM:

1. Outperformed both baselines by up to 6× and 11×, which leads to 9× and 16× fewer SLO violations;
2. Lowered the overall requested CPU limit by 29–62%, as shown in Fig. 11(b), and increased the average cluster-level CPU utilization by up to 33%; and
3. Reduced the number of dropped or timed out user requests by up to 8× as shown in Fig. 11(c).

FIRM can provide these benefits because it detects SLO violations accurately and addresses resource contention before SLO violations can propagate. By interacting with dynamic microservice environments under complicated loads and resource allocation scenarios, FIRM's RL agent dynamically learns the policy, and hence outperforms heuristics-based approaches.

5 Discussion

Necessity and Challenges of Modeling Low-level Resources. Recall from §2 that modeling of resources at a fine granularity is necessary, as it allows for better performance without overprovisioning. It is difficult to model the dependence between low-level resource requirements and quantifiable performance gain while dealing with uncertain and noisy measurements [76, 120]. FIRM addresses the issue by modeling that dependency in an RL-based feedback loop, which automatically explores the action space to generate optimal policies without human intervention.

Why a Multilevel ML Framework? A model of the states of all microservices that is fed as the input to a single large ML model [81, 126] leads to (i) state-action space explosion issues that grow with the number of microservices, thus increasing the training time; and (ii) dependence between the microservice architecture and the ML-model, which sacrifices the generality. FIRM addresses those problems by incorporating a two-level ML framework. The first level ML model uses SVM to filter the microservice instances responsible for SLO violations, thereby reducing the number of microservices that need to be considered in mitigating SLO violations. That enables the second level ML model, the RL agent, to be trained faster and removes dependence on the application architecture. That, in turn, helps avoid RL model reconstruction/retraining.

Table 6: Avg. latency for resource management operations.

Operation	Partition (Scale Up/Down)					Container Start	
	CPU	Mem	LLC	I/O	Net	Warm	Cold
Mean (*ms*)	2.1	42.4	39.8	2.3	12.3	45.7	2050.8
Std Dev (*ms*)	0.3	11.0	9.2	0.4	1.1	6.9	291.4

Lower Bounds on Manageable SLO Violation Duration for FIRM. As shown in Table 6, the operations to scale resources for microservice instances take 2.1–45.7 ms. Thus, that is the minimum duration of latency spikes that any RM approach can handle. For transient SLO violations, which last shorter than the minimum duration, the action generated by FIRM will always miss the mitigation deadline and can potentially harm overall system performance. Worse, it may lead to oscillations between scaling operations. Predicting the spikes before they happen, and proactively taking mitigation actions can be a solution. However, it is a generally-acknowledged difficult problem, as microservices are dynamically evolving, in terms of both load and architectural design, which is subject to our future work.

Limitations. FIRM has several limitations that we plan to address in future work. First, FIRM currently focuses on resource interference caused by real workload demands. However, FIRM lacks the ability to detect application bugs or misconfigurations, which may lead to failures such as memory leak. Allocating more resources to such microservice instances may harm the overall resource efficiency. Other sources of SLO violations, including global resource sharing (e.g., network switches or global file systems) and hardware causes (e.g., power-saving energy management), are also beyond FIRM's scope. Second, the scalability of FIRM is limited by the maximum scalability of the centralized graph database, and the boundary caused by the network traffic telemetry overhead. (Recall the lower bound on the SLO violation duration.) Third, we plan to implement FIRM's tracing module based on side-car proxies (i.e., service meshes) [15] that minimizes application instrumentation and has wider support of programming languages.

6 Related Work

SLO violations in cloud applications and microservices are a popular and well-researched topic. We categorize prior work into two buckets: root cause analyzers and autoscalers. Both rely heavily on the collection of tracing and telemetry data.

Tracing and Probing for Microservices. Tracing for large-scale microservices (essentially distributed systems) helps understand the path of a request as it propagates through the components of a distributed system. Tracing requires either application-level instrumentation [18,32,57,95,111–115] or middleware/OS-level instrumentation [10,16,63,109] (e.g., Sieve [109] utilizes a kernel module *sysdig* [103] which provides system calls as an event stream containing tracing information about the monitored process to a user application).

Root Cause Analysis. A large body of work [16, 35, 50, 52, 61, 63, 93, 109, 121, 124] provides promising evidence that data-driven diagnostics help detect performance anomalies and analyze root causes. For example, Sieve [109] leverages Granger causality to correlate performance anomaly data series with particular metrics as potential root causes. Pinpoint [16] runs clustering analysis on Jaccard similarity coefficient to determine the components that are mostly correlated with the failure. Microscope [61] and MicroRCA [124] are both designed to identify abnormal services by constructing service causal graphs that model anomaly propagation and by inferring causes using graph traversal or ranking algorithms [51]. Seer [35] uses deep learning to learn spatial and temporal patterns that translate to SLO violations. However, none of these approaches addresses the dynamic nature of microservice environments (i.e., frequent microservice updates and deployment changes), which require costly model reconstruction or retraining.

Autoscaling Cloud Applications. Current techniques for autoscaling cloud applications can be categorized into four groups [65,84]: (a) rule-based (commonly offered by cloud providers [6, 7, 37]), (b) time series analysis (regression on resource utilization, performance, and workloads), (c) model-based (e.g., queueing networks), or (d) RL-based. Some approaches combine several techniques. For instance, Auto-pilot [88] combines time series analysis and RL algorithms to scale the number of containers and associated CPU/RAM. Unfortunately, when applied to microservices with large scale and complex dependencies, independent scaling of each microservice instance results in suboptimal solutions (because of critical path intersection and insight 2 in §2), and it is difficult to define sub-SLOs for individual instances. Approaches for autoscaling microservices or distributed dataflows [39,56,81,126,127] make scaling decisions on the number of replicas and/or container size without considering low-level shared-resource interference. ATOM [39] and Microscaler [127] do so by using a combination of queueing network- and heuristic-based approximations. ASFM [81] uses recurrent neural network activity to predict workloads and translates application performance to resources by using linear regression. Streaming and data-processing scalers like DS2 [56] and MIRAS [126] leverage explicit application-level modeling and apply RL to represent the resource-performance mapping of operators and their dependencies.

Cluster Management. The emergence of cloud computing motivates the prevalence of cloud management platforms that provide services such as monitoring, security, fault tolerance, and performance predictability. Examples include Borg [119], Mesos [43], Tarcil [28], Paragon [25], Quasar [26], Morpheus [54], DeepDive [73], and Q-clouds [71]. In this paper, we do not address the problem of cluster orchestration. FIRM can work in conjunction with those cluster management tools to reduce SLO violations.

7 Conclusion

We propose *FIRM*, an ML-based, fine-grained resource management framework that addresses SLO violations and resource underutilization in microservices. FIRM uses a two-level ML model, one for identifying microservices responsible for SLO violations, and the other for mitigation. The combined ML model reduces SLO violations up to 16× while reducing the overall CPU limit by up to 62%. Overall, FIRM enables fast mitigation of SLOs by using efficient resource provisioning, which benefits both cloud service providers and microservice owners. FIRM is open-sourced at https://gitlab.engr.illinois.edu/DEPEND/firm.git.

8 Acknowledgment

We thank the OSDI reviewers and our shepherd, Rebecca Isaacs, for their valuable comments that improved the paper. We appreciate K. Atchley, F. Rigberg, and J. Applequist for their insightful comments on the early drafts of this manuscript. This research was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under award No. 2015-02674. This work is partially supported by the National Science Foundation (NSF) under grant No. 2029049; by a Sandia National Laboratories[6] under contract No. 1951381; by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration that is part of the IBM AI Horizon Network; and by Intel and NVIDIA through equipment donations. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, Sandia National Laboratories, IBM, NVIDIA, or, Intel. Saurabh Jha is supported by a 2020 IBM PhD fellowship.

[6]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] Ivo Adan and Jacques Resing. *Queueing theory*. Eindhoven University of Technology Eindhoven, 2002.

[2] Bernhard Ager, Fabian Schneider, Juhoon Kim, and Anja Feldmann. Revisiting cacheability in times of user generated content. In *2010 INFOCOM IEEE Conference on Computer Communications Workshops*, pages 1–6. IEEE, 2010.

[3] Younsun Ahn, Jieun Choi, Sol Jeong, and Yoonhee Kim. Auto-scaling method in hybrid cloud for scientific applications. In *Proceedings of the 16th Asia-Pacific Network Operations and Management Symposium*, pages 1–4. IEEE, 2014.

[4] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of The 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 103–112, 2014.

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[6] AWS auto scaling documentation. https://docs.aws.amazon.com/autoscaling/index.html, Accessed 2020/01/23.

[7] Azure autoscale. https://azure.microsoft.com/en-us/features/autoscale/, Accessed 2020/01/23.

[8] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: An experience report. In *Proceedings of the European Conference on Service-Oriented and Cloud Computing*, pages 201–215. Springer, 2015.

[9] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[10] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[11] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[12] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise Reduction in Speech Processing*, pages 1–4. Springer, 2009.

[13] cAdvisor. https://github.com/google/cadvisor, Accessed 2020/01/23.

[14] Luiz A. Celiberto Jr, Jackson P. Matsuura, Ramón López De Màntaras, and Reinaldo A.C. Bianchi. Using transfer learning to speed-up reinforcement learning: A case-based approach. In *Proceedings of 2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*, pages 55–60. IEEE, 2010.

[15] Ramaswamy Chandramouli and Zack Butcher. Building secure microservices-based applications using service-mesh architecture. *NIST Special Publication*, 800:204A, 2020.

[16] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.

[17] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and Jose F. Martinez. Workload characterization of interactive cloud services on big and small server platforms. In *Proceedings of 2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 125–134. IEEE, 2017.

[18] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, 2014.

[19] Docker Swarm. https://www.docker.com/products/docker-swarm, Accessed 2020/01/23.

[20] Kubernetes. https://kubernetes.io/, Accessed 2020/01/23.

[21] CoreOS rkt, a security-minded, standards-based container engine. https://coreos.com/rkt/, Accessed 2020/01/23.

[22] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[23] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Resource provisioning of web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, pages 49–60. USENIX Association, 2011.

[24] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying interference for datacenter applications. In *Proceedings of 2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 23–33. IEEE, 2013.

[25] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[26] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[27] Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. *Communications of the ACM*, 61(8):65–72, 2018.

[28] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110, 2015.

[29] Christopher P. Diehl and Gert Cauwenberghs. SVM incremental learning, adaptation and optimization. In *Proceedings of 2003 International Joint Conference on Neural Networks*, volume 4, pages 2685–2690. IEEE, 2003.

[30] Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, Nathaniel Morris, and Christopher Stewart. Characterizing service level objectives for cloud services: Realities and myths. In *Proceedings of 2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 200–206. IEEE, 2019.

[31] Rob Eisinga, Manfred Te Grotenhuis, and Ben Pelzer. The reliability of a two-item scale: Pearson, Cronbach, or Spearman-Brown? *International Journal of Public Health*, 58(4):637–642, 2013.

[32] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, pages 271–284, 2007.

[33] Yu Gan and Christina Delimitrou. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018.

[34] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[35] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 19–33, 2019.

[36] Mrittika Ganguli, Rajneesh Bhardwaj, Ananth Sankaranarayanan, Sunil Raghavan, Subramony Sesha, Gilbert Hyatt, Muralidharan Sundararajan, Arkadiusz Chylinski, and Alok Prakash. CPU overprovisioning and cloud compute workload scheduling mechanism, March 20 2018. US Patent 9,921,866.

[37] Google cloud load balancing and scaling. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling, Accessed 2020/01/23.

[38] Panos Gevros and Jon Crowcroft. Distributed resource management with heterogeneous linear controls. *Computer Networks*, 45(6):835–858, 2004.

[39] Alim Ul Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven autoscaling for microservices. In *Proceedings of 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004. IEEE, 2019.

[40] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of 2007 IEEE 10th International Symposium on Workload Characterization*, pages 171–180. IEEE, 2007.

[41] Ivo Grondman, Lucian Busoniu, Gabriel A.D. Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307, 2012.

[42] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[43] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 295–208, 2011.

[44] Todd Hoff. Latency is everywhere and it costs you sales: How to crush it, July 2009. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, Accessed 2020/01/23.

[45] HTB - Hierarchical Token Bucket. https://linux.die.net/man/8/tc-htb, Accessed 2020/01/23.

[46] Steven Ihde and Karan Parikh. From a monolith to microservices + REST: The evolution of LinkedIn's service architecture, March 2015. https://www.infoq.com/presentations/linkedin-microservices-urn/, Accessed 2020/01/23.

[47] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro V. Papadopoulos, Bogdan Ghit, Dick Epema, and Alexandru Iosup. An experimental performance evaluation of autoscaling policies for complex workflows. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 75–86, 2017.

[48] Intel cache allocation technology. https://github.com/intel/intel-cmt-cat, Accessed 2020/01/23.

[49] Intel memory bandwidth allocation. https://github.com/intel/intel-cmt-cat, Accessed 2020/01/23.

[50] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Performance monitoring and root cause analysis for cloud-hosted web applications. In *Proceedings of the 26th International Conference on World Wide Web*, pages 469–478, 2017.

[51] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*, pages 271–279, 2003.

[52] Saurabh Jha, Shengkun Cui, Subho Banerjee, Tianyin Xu, Jeremy Enos, Mike Showerman, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Live forensics for HPC systems: A case study on distributed storage systems. In *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis*, 2020.

[53] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32, 2019.

[54] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, 2016.

[55] Autoscaling in Kubernetes. https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/, Accessed 2020/01/23.

[56] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.

[57] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 34–50, 2017.

[58] Pavel Laskov, Christian Gehl, Stefan Krüger, and Klaus-Robert Müller. Incremental support vector learning: Analysis, implementation and applications. *Journal of Machine Learning Research*, 7(Sep):1909–1936, 2006.

[59] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[60] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of 2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[61] Jinjin Lin, Pengfei Chen, and Zibin Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *Proceedings of International Conference on Service-Oriented Computing*, pages 3–20. Springer, 2018.

[62] Richard Harold Lindeman. Introduction to bivariate and multivariate analysis. Technical report, Scott Foresman & Co, 1980.

[63] Haifeng Liu, Jinjun Zhang, Huasong Shan, Min Li, Yuan Chen, Xiaofeng He, and Xiaowei Li. JCallGraph: Tracing microservices in very large scale container cloud platforms. In *Proceedings of International Conference on Cloud Computing*, pages 287–302. Springer, 2019.

[64] Keith Gerald Lockyer. *Introduction to Critical Path Analysis*. Pitman, 1969.

[65] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[66] Michael David Marr and Matthew D. Klein. Automated profiling of resource usage, April 26 2016. US Patent 9,323,577.

[67] Jason Mars and Lingjia Tang. Whare-Map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 619–630, 2013.

[68] Tony Mauro. Adopting microservices at Netflix: Lessons for architectural design, February 2015. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, Accessed 2020/01/23.

[69] Matt McGee. It's official: Google now counts site speed as a ranking factor, April 2010. https://searchengineland.com/google-now-counts-site-speed-as-ranking-factor-39708, Accessed 2020/01/23.

[70] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2–2, 2014.

[71] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, pages 237–250, 2010.

[72] Neo4j: Native Graph Database. https://github.com/neo4j/neo4j, Accessed 2020/01/23.

[73] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of 2013 USENIX Annual Technical Conference (USENIX ATC)*, pages 219–230, 2013.

[74] NumPy. https://numpy.org/doc/stable/index.html, Accessed 2020/01/23.

[75] OpenTracing. https://opentracing.io/, Accessed 2020/01/23.

[76] Karl Ott and Rabi Mahapatra. Hardware performance counters for embedded software anomaly detection. In *Proceedings of 2018 IEEE 16th Intl. Conf. on Dependable, Autonomic and Secure Computing, the 16th Intl. Conf. on Pervasive Intelligence and Computing, the 4th Intl. Conf. on Big Data Intelligence and Computing and Cyber Science and Technology Congress*, pages 528–535. IEEE, 2018.

[77] Dan Paik. Adapt or Die: A microservices story at Google, December 2016. https://www.slideshare.net/apigee/adapt-or-die-a-microservices-story-at-google, Accessed 2020/01/23.

[78] Panagiotis Patros, Stephen A. MacKay, Kenneth B. Kent, and Michael Dawson. Investigating resource interference and scaling on multitenant PaaS clouds. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pages 166–177, 2016.

[79] perf. http://man7.org/linux/man-pages/man1/perf.1.html, Accessed 2020/01/23.

[80] pmbw: Parallel Memory Bandwidth Benchmark. https://panthema.net/2013/pmbw/, Accessed 2020/01/23.

[81] Issaret Prachitmutita, Wachirawit Aittinonmongkol, Nasoret Pojjanasuksakul, Montri Supattatham, and Praisan Padungweang. Auto-scaling microservices on IaaS under SLA with cost-effective framework. In *Proceedings of 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, pages 583–588. IEEE, 2018.

[82] The Prometheus monitoring system and time series database. https://github.com/prometheus/prometheus, Accessed 2020/01/23.

[83] PyTorch. https://pytorch.org/, Accessed 2020/01/23.

[84] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):1–33, 2018.

[85] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 2—-12. Association for Computing Machinery, 2006.

[86] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review*, 37(4):337–348, 2007.

[87] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC 12)*, pages 1–13, 2012.

[88] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[89] Cristian Satnic. Amazon, Microservices and the birth of AWS cloud computing, April 2016. https://www.linkedin.com/pulse/amazon-microservices-birth-aws-cloud-computing-cristian-satnic/, Accessed 2020/01/23.

[90] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[91] scikit-learn. https://scikit-learn.org/stable/, Accessed 2020/01/23.

[92] S. Senthil Kumaran. *Practical LXC and LXD: Linux Containers for Virtualization and Orchestration*. Springer, 2017.

[93] Syed Yousaf Shah, Xuan-Hong Dang, and Petros Zerfos. Root cause detection using dynamic dependency graphs from time series data. In *Proceedings of 2018 IEEE International Conference on Big Data (Big Data)*, pages 1998–2003. IEEE, 2018.

[94] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Proceedings of 2011 31st International Conference on Distributed Computing Systems*, pages 559–570. IEEE, 2011.

[95] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[96] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 733–750, 2020.

[97] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. SoftSKU: optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[98] Akshitha Sriraman and Thomas F. Wenisch. μsuite: a benchmark suite for microservices. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[99] Akshitha Sriraman and Thomas F. Wenisch. μtune: Auto-tuned threading for OLDI microservices. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, 2018.

[100] stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng, Accessed 2020/01/23.

[101] Sonja Stüdli, M. Corless, Richard H. Middleton, and Robert Shorten. On the modified AIMD algorithm for distributed resource management with saturation of each user's share. In *Proceedings of 2015 54th IEEE Conference on Decision and Control (CDC)*, pages 1631–1636. IEEE, 2015.

[102] Sysbench. https://github.com/akopytov/sysbench, Accessed 2020/01/23.

[103] Sysdig. https://sysdig.com/, Accessed 2020/01/23.

[104] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[105] Matthew E. Taylor, Gregory Kuhlmann, and Peter Stone. Autonomous transfer for reinforcement learning. In *Proceedings of 2008 International Conference of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 283–290, 2008.

[106] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, pages 1633–1685, Jul 2009.

[107] tc: Traffic Control in the Linux kernel. https://linux.die.net/man/8/tc, Accessed 2020/01/23.

[108] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 199–212, 2018.

[109] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 14–27, 2017.

[110] Scott Tonidandel and James M. LeBreton. Relative importance analysis: A useful supplement to regression analysis. *Journal of Business and Psychology*, 26(1):1–9, 2011.

[111] Instana. https://docs.instana.io/, Accessed 2020/01/23.

[112] Jaeger: Open source, end-to-end distributed tracing. https://jaegertracing.io/, Accessed 2020/01/23.

[113] Lightstep distributed tracing. https://lightstep.com/distributed-tracing/, Accessed 2020/01/23.

[114] SkyWalking: An application performance monitoring system. https://github.com/apache/skywalking, Accessed 2020/01/23.

[115] OpenZipkin: A distributed tracing system. https://zipkin.io/, Accessed 2020/01/23.

[116] Train-Ticket: A train-ticket booking system based on microservice architecture. https://github.com/FudanSELab/train-ticket, Accessed 2020/01/23.

[117] Trickle: A lightweight userspace bandwidth shaper. https://linux.die.net/man/1/trickle, Accessed 2020/01/23.

[118] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *Proceedings of 2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[119] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 1–17, Bordeaux, France, 2015.

[120] Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–23, 2016.

[121] Jianping Weng, Jessie Hui Wang, Jiahai Yang, and Yang Yang. Root cause analysis of anomalies of multitier services in public clouds. *IEEE/ACM Transactions on Networking*, 26(4):1646–1659, 2018.

[122] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 266–275, 2003.

[123] wrk2: An HTTP benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2, Accessed 2020/01/23.

[124] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. MicroRCA: Root cause localization of performance issues in microservices. In *Proceedings of 2020 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 1–9, 2020.

[125] Cui-Qing Yang and Barton Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 366–367, 1988.

[126] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *Proceedings of 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 122–132. IEEE, 2019.

[127] Guangba Yu, Pengfei Chen, and Zibin Zheng. Microscaler: Automatic scaling for microservices with an online learning approach. In *Proceedings of 2019 IEEE International Conference on Web Services (ICWS)*, pages 68–75. IEEE, 2019.

[128] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 14(8):1–1, 2018.

A Artifact Appendix

A.1 Abstract

FIRM is publicly available at https://gitlab.engr.illinois.edu/DEPEND/firm.git. We provide implementations for FIRM's SVM-based critical component extraction, RL-based SLO violation mitigation, and the performance anomaly injection. In addition, we provide a tracing data set of the four microservice benchmarks deployed on our dedicated Kubernetes cluster of 15 physical nodes. The data set was generated by running open-loop workload generation and performance anomaly injection.

A.2 Artifact Check-list

- **Algorithm:** FIRM's critical component extraction includes an algorithm to find the weighted longest path (i.e., critical path analysis) from the execution history graph of microservices.
- **Model:** FIRM's two-level machine learning architecture includes an SVM-based critical component extraction model and an RL-based SLO violation mitigation model. The latter one is designed based on deep deterministic policy gradient (DDPG).
- **Data set:** The artifact includes a tracing data set collected by running four microservice benchmarks [34, 116] in a 15-node Kubernetes cluster. The microservice benchmarks are driven by workload generation and performance anomaly injection.
- **Hardware:** Experiments can run on a cluster of physical nodes with Intel Cache Allocation Technology (CAT) [48] and Intel Memory Bandwidth Allocation (MBA) [49] enabled.
- **Required disk space:** Neo4j [72] requires 10 GB minimum block storage, and the storage size depends on the size of the database.
- **Set-up instructions:** Set-up instructions are available at the `README.md` file in the repository.
- **Public link:** https://gitlab.engr.illinois.edu/DEPEND/firm.git
- **Code licenses:** Apache License Version 2.0
- **Data licenses:** CC0 License

A.3 Description

A.3.1 How to Access

The artifact is publicly available at https://gitlab.engr.illinois.edu/DEPEND/firm.git.

A.3.2 Hardware Dependencies

Experiments can be run on a cluster of physical nodes with processors that have Intel CAT and MBA technologies enabled. They are required for last-level cache partitioning and memory bandwidth partitioning respectively.

A.3.3 Software Dependencies

Software dependencies are specified at the `README.md` file, which includes Kubernetes, Docker-Compose, and Docker.

A.3.4 Data Sets

The tracing data sets of four microservice benchmarks deployed on our dedicated Kubernetes cluster consisting of 15 heterogeneous nodes are also available. The data sets are not sampled and are from selected types of requests in each benchmark, i.e., compose-posts in the social network application, compose-reviews in the media service application, book-rooms in the hotel reservation application, and reserve-tickets in the train ticket booking application. A detailed description is available at `data/README.md`.

A.4 Installation

Installation instructions are specified at the `README.md` file in the repository.

A.5 Experiment Workflow

Experiments on physical clusters start from deploying the Kubernetes with FIRM. Microservice applications instrumented with the OpenTracing [75] standard are then deployed in the Kubernetes cluster. One can also use the instrumented microservice benchmarks in the repository for experiments. To drive the experiments, workload generators and performance anomaly injectors should be configured and installed accordingly. Then the training of FIRM's ML models is divided into two phases. In the first phase, the workflow stops at the SLO violation localization. The SVM model is trained with the feature data retrieved from the tracing coordinator and the label data from the performance anomaly injection campaign. In the second phase, the workflow continues and FIRM's RL agent is trained by interacting with the environment.

A.6 Experiment Customization

FIRM's multilevel ML modeling provides the flexibility of customizing the algorithms for both SLO violation localization and mitigation. The SVM model can be replaced by other supervised learning models or other heuristics-based methods. The DDPG algorithm used by the RL agent can also be replaced by other RL algorithms. The repository consists of the implementations of other alternative RL models such as proximal policy optimization (PPO) and policy gradient.

In addition, different types of resources in control are also configurable in the RL agent and the performance anomaly injector. That pluggability allows one to add or remove resources, and to change the actions associated with each type of resource.

A.7 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts



Building Scalable and Flexible Cluster Managers Using Declarative Programming

Lalith Suresh, João Loff[1], Faria Kalim[2], Sangeetha Abdu Jyothi[3], Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, Michael Gasch
VMware, [1]IST (ULisboa) / INESC-ID, [2]UIUC, [3]UC Irvine and VMware

Abstract

Cluster managers like Kubernetes and OpenStack are notoriously hard to develop, given that they routinely grapple with hard combinatorial optimization problems like load balancing, placement, scheduling, and configuration. Today, cluster manager developers tackle these problems by developing system-specific best effort heuristics, which achieve scalability by significantly sacrificing the cluster manager's decision quality, feature set, and extensibility over time. This is proving untenable, as solutions for cluster management problems are routinely developed from scratch in the industry to solve largely similar problems across different settings.

We propose DCM, a radically different architecture where developers specify the cluster manager's behavior declaratively, using SQL queries over cluster state stored in a relational database. From the SQL specification, the DCM compiler synthesizes a program that, at runtime, can be invoked to compute policy-compliant cluster management decisions given the latest cluster state. Under the covers, the generated program efficiently encodes the cluster state as an optimization problem that can be solved using off-the-shelf solvers, freeing developers from having to design ad-hoc heuristics.

We show that DCM significantly lowers the barrier to building scalable and extensible cluster managers. We validate our claim by powering three production-grade systems with it: a Kubernetes scheduler, a virtual machine management solution, and a distributed transactional datastore.

1 Introduction

Today's data centers are powered by a variety of cluster managers like Kubernetes [10], DRS [47], Openstack [15], and OpenShift [14]. These systems configure large-scale clusters and allocate resources to jobs. Whether juggling containers, virtual machines, micro-services, virtual network appliances, or serverless functions, these systems must enforce numerous cluster management *policies*. Some policies represent *hard constraints*, which must hold in any valid system configuration; e.g., "each container must obtain its minimal requested amount of disk space". Others are *soft constraints*, which reflect preferences and quality metrics; e.g., "prefer to scatter replicas across as many racks as possible". A cluster manager therefore solves a challenging combinatorial optimization problem of finding configurations that satisfy hard constraints while minimizing violations of soft constraints.

Despite the complexity of the largely similar algorithmic problems involved, cluster managers in various contexts tackle the configuration problem using custom, system-specific best-effort heuristics—*an approach that often leads to a software engineering dead-end* (§2). As new types of policies are introduced, developers are overwhelmed by having to write code to solve arbitrary combinations of increasingly complex constraints. This is unsurprising given that most cluster management problems involve *NP-hard combinatorial optimization* that cannot be efficiently solved via naive heuristics. Besides the algorithmic complexity, the lack of separation between the cluster state, the constraints, and the constraint-solving algorithm leads to high code complexity and maintainability challenges, and hinders re-use of cluster manager code across different settings (§2). In practice, even at a large software vendor we find policy-level feature additions to cluster managers *take months* to develop.

Our contribution This paper presents *Declarative Cluster Managers* (DCM), a radically different approach to building cluster managers, wherein the implementation to compute policy-compliant configurations is *synthesized* by a compiler from a high-level specification.

Specifically, developers using DCM maintain cluster state in a relational database, and *declaratively* specify the constraints that the cluster manager should enforce using SQL. Given this specification, DCM's compiler synthesizes a program that, at runtime, can be invoked to pull the latest cluster state from the database and compute a set of policy-compliant changes to make to that state (e.g., compute optimal placement decisions for newly launched virtual machines). The generated program – an *encoder* – encodes the cluster state and constraints into an optimization model that is then solved using a constraint solver.

In doing so, DCM significantly lowers the barrier to building cluster managers that achieve all three of *scalability*, *high decision quality*, and *extensibility* to add new features and policies. In contrast, today's cluster managers use custom heuristics that heavily sacrifice both decision quality and extensibility to meet scalability goals (§2).

For scalability, our compiler generates implementations that construct highly efficient constraint solver encodings that scale to problem sizes in large clusters (e.g., 53% improved p99 placement latency in a 500 node cluster over the heavily optimized Kubernetes scheduler, §6.1).

For high decision quality, the use of constraint solvers under-the-covers guarantees optimal solutions for the specified problems, with the freedom to relax the solution quality if needed (e.g., 4× better load balancing in a commercial virtual machine resource manager, §6.2).

For extensibility, DCM enforces a strict separation between the a) cluster state, b) the modular and concise representation of constraints in SQL, and c) the solving logic. This makes it easy to add new constraints and non-trivial features (e.g., making a Kubernetes scheduler place both pods and virtual machines in a custom Kubernetes distribution, §6.3).

Several research systems [46, 53, 57, 78, 92] propose to use constraint solvers for cluster management tasks. These systems all involve a significant amount of effort from optimization experts to handcraft an encoder for specific problems with simple, well-defined constraints – let alone encode the full complexity and feature sets of production-grade cluster managers (e.g., Kubernetes has 30 policies for driving initial placement alone). Even simple encoders are challenging to scale to large problem sizes and are not extensible even when they do scale (§8). In fact, for these reasons, *constraint solvers remain rarely used* within production-grade cluster managers in the industry-at-large: none of the open-source cluster managers use solvers and, anecdotally, nor do widely used enterprise offerings in this space.

Instead, with DCM, developers need not handcraft heuristics nor solver encodings to tackle challenging cluster management problems.

Providing a capability like DCM is fraught with challenges. First, cluster managers operate in a variety of modes and timescales: from incrementally placing new workloads at millisecond timescales, to periodically performing global reconfiguration (like load balancing or descheduling); we design a programming model that is flexible enough to accommodate these various use cases within a single system (§3). Second, constraint solvers are not a panacea and are notoriously hard to scale to large problem sizes. DCM's compiler uses carefully designed optimization passes that bridge the wide chasm between a high-level SQL specification of a cluster management problem and an efficient, low-level representation of an optimization model – doing so leverages the strengths of the constraint solver while avoiding its weaknesses (§4).

Summary of results We report in-depth about our experience building and extending a Kubernetes Scheduler using DCM. We implement existing policies in Kubernetes in under 20 lines of SQL each. On a 500 node Kubernetes cluster on AWS, DCM improves 95^{th} percentile pod placement latencies by up to 2×, is 10× more likely to find feasible placements in constrained scenarios, and correctly preempts pods 2× faster than the baseline scheduler. We also report simulation results with up to 10K node clusters and experiment with non-trivial extensions to the scheduler, like placing both pods and VMs within a custom Kubernetes distribution. We also use DCM to power a commercial virtual machine management solution where we improved load balancing quality by 4×. Lastly, we briefly discuss a distributed transactional datastore where we implemented several features with a few lines of SQL.

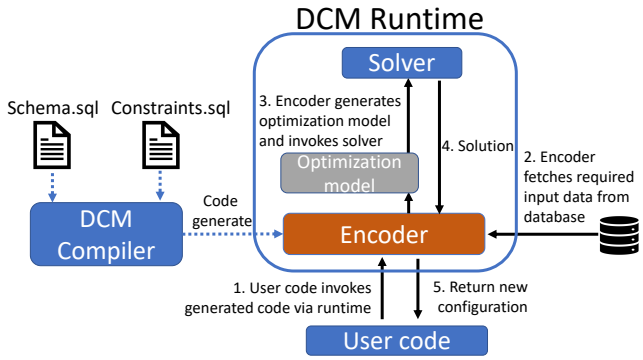


Figure 1: DCM architecture. Dotted lines show the compilation flow. Solid lines show runtime interactions between the DCM runtime, user code and the cluster state DB.

2 Motivation

Our motivating concern is that ad-hoc solutions for cluster management problems are regularly built from scratch in the industry, due to the wide range of specialized data-center environments and workloads that organizations have, for which off-the-shelf solutions do not suffice. Even beyond dedicated cluster managers like Kubernetes [10], OpenStack [15], and Nomad [50], similar capabilities are routinely embedded within enterprise-grade distributed systems like databases and storage systems: e.g., for policy-based configuration, data replication, or load-balancing across machines, all of which are combinatorial optimization problems.

Today, developers handcraft heuristics to solve these cluster management problems that incur significant engineering overhead. First, the heuristics are hard to scale to clusters with hundreds to thousands of nodes; they often require purpose-built and inflexible pre-computing and caching optimizations to remain tractable [40, 95]. Even then, the heuristics are challenging to get right as developers have to account for arbitrary combinations of constraints. Second, the heuristics sacrifice decision quality to scale (e.g., load balancing quality), which is not surprising given that combinatorial optimization problems cannot be solved efficiently via naive heuristics. Third, they lead to complex code that makes it hard to extend and evolve the cluster manager over time; it is not uncommon for policy-level feature additions to take multiple months' worth of effort to deliver.

We illustrate the above challenges using Kubernetes as a representative example.

Kubernetes example The Kubernetes Scheduler is responsible for assigning groups of containers, called *pods*, to cluster

Policy	Description
H1-4	Avoid nodes with resource overload, unavailability or errors
H5	Resource capacity constraints: pods scheduled on a node must not exceed node's CPU, memory, and disk capacity
H6	Ensure network ports on host machine requested by pod are available
H7	Respect requests by a pod for specific nodes
H8	If pod is already assigned to a node, do not reassign
H9	Ensure pod is in the same zone as its requested volumes
H10-11	If a node has a 'taint' label, ensure pods on that node are configured to tolerate those taints
H12-13	Node affinity/anti-affinity: pods are affine/anti-affine to nodes according to configured labels
H14	Inter-pod affinity/anti-affinity: pods are affine/anti-affine to each other according to configured labels
H15	Pods of the same service must be in the same failure-domain
H16-20	Volume constraints specific to GCE, AWS, Azure.
S1	Spread pods from the same group across nodes
S2-5	Load balance pods according to CPU/Memory load on nodes
S6	Prefer nodes that have matching labels
S7	Inter-pod affinity/anti-affinity by labels
S8	Prefer to not exceed node resource limits
S9	Prefer nodes where container images are already available

Figure 2: Policies from the baseline Kubernetes scheduler, showing both hard (H) constraints and soft (S) constraints.

nodes (physical or virtual machines). Each pod has a number of user-supplied attributes describing its resource demand (CPU, memory, storage, and custom resources) and placement preferences (the pod's affinity or anti-affinity to other pods or nodes). These attributes represent hard constraints that must be satisfied for the pod to be placed on a node (H1–H20 in Table 2). Kubernetes also supports soft versions of placement constraints, with a violation cost assigned to each constraint (S1–S9 in Table 2). Like other task-by-task schedulers [15, 94, 95], the Kubernetes default scheduler uses a greedy, best-effort heuristic to place one task (pod) at a time, drawn from a queue. For each pod, the scheduler tries to find feasible nodes according to the hard constraints, score them according to the soft constraints, and pick the best-scored node. Feasibility and scoring are parallelized for speed.

Decision quality: not guaranteed to find feasible, let alone optimal, placements Pod scheduling is a variant of the multidimensional bin packing problem [18, 21], which is NP-hard and cannot, in the general case, be solved efficiently with greedy algorithms. This is especially the case when the scheduling problem is *tight* due to workload consolidation and users increasingly relying on affinity constraints for performance and availability.

To remain performant, the Kubernetes scheduler only considers a random subset of nodes when scheduling a pod, which might miss feasible nodes [93]. Furthermore, the scheduler may commit to placing a pod and deny feasible choices from pods that are already pending in the scheduler's queue (a common weakness among task-by-task schedulers [40]).

Feature limitations: best-effort scheduling does not support global reconfiguration Many scenarios require the scheduler to simultaneously reconfigure arbitrary groups of pods

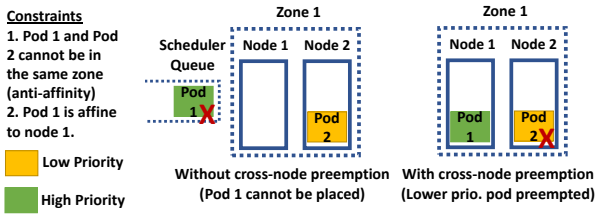


Figure 3: An anti-affinity constraint prevents Pod 1 and Pod 2 from being in the same zone, pod 1 is affine to node 1, and pod 2 has a lower priority than pod 1. Placing pod 1 on node 1 requires evicting pod 2 on node 2.

and nodes. For instance, Figure 3 shows a scenario where a high priority pod (pod 1) can only be placed on node 1, but to do so, the scheduler has to preempt a lower priority pod on node 2. Computing this rearrangement requires simultaneous reasoning about resource and affinity constraints spanning multiple pods and nodes, which cannot be achieved in the current architecture. Thus, although such global reconfiguration is in high demand among users, it is unsupported in Kubernetes [60, 64].

Extensibility: Best-effort scheduling leads to complex code Similar to Borg [40, 95], Kubernetes needs careful engineering to keep scheduling tractable at scale. Several policies like inter-pod affinity (Table 2-H14) and service affinities (Table 2-H15) are compute intensive because they require reasoning over groups of pods. These policies are kept tractable using carefully designed caching and pre-computing optimizations that are fragile in the face of evolving requirements. For example, it is hard to extend inter-pod affinity policies to specify the number of pods per node [58, 59, 61–63], and there are discussions among developers to restrict these policies to make the code efficient [60]. For similar reasons, there are discussions among developers to remove the service affinity policy due to accumulating technical debt around its pre-computing optimizations [69]. Such complexity accumulates to make entire classes of policies requested by users difficult to implement in the scheduler [60, 64, 73].

Beyond policy-level extensions, the tight coupling between the cluster state representation in the scheduler's data-structures and the scheduling logic makes it near impossible to introduce changes to the underlying abstractions (e.g., extending the scheduler to also place tasks other than pods, like virtual machines [71]) without a complete rewrite [66].

3 Declarative Programming with DCM

Our position is that developers should specify cluster management policies using a high-level declarative language, and let an automated tool like DCM generate the logic that efficiently computes policy-compliant decisions. Architecturally,

it allows the behavior of the cluster manager to be described and evolved independently of the implementation details.

We use SQL as the declarative language for specifying policies for multiple reasons. First, it allows us to consistently describe and manipulate both the cluster state and the constraints on that state. Second, it is a battle-tested and widely known language, which aids adoption. Third, it is sufficiently expressive that we have not felt the need for designing yet another DSL (§4.1.1).

We now demonstrate DCM's capabilities and programming model with a simplified guide to building a Kubernetes scheduler with it. Our scheduler operates as a drop-in replacement for the default scheduler (§2), supporting all of its capabilities and adding new ones.

The workflow in a DCM-powered scheduler consists of three steps (Figure 1). First, the scheduler stores the cluster state in an SQL database based on an SQL schema designed by the developer. Second, the developer extends the schema with scheduling constraints, also written in SQL. The compiler generates an encoder based on the constraints and schema. Third, at runtime, the scheduler invokes the generated encoder via the DCM library as new pods are added to the system. The generated encoder pulls the required cluster state from the database, produces and solves an optimization model that is parameterized by that state, and outputs the required scheduling decisions.

Cluster state database Kubernetes stores all state (of nodes and pods) in an etcd [36] cluster. The default scheduler maintains a cache of relevant parts of this state locally using in-memory data structures. In DCM, we replace this cache with an in-memory embedded SQL database (H2 [4]) and specify an SQL schema (tables and views) to represent the cluster state. Currently, the schema uses 18 tables and 12 views to describe the set of pods, nodes, volumes, container images, pod labels, node labels, and other cluster state. The developer annotates some columns in the schema as *decision variables*, i.e., variables to be assigned automatically by DCM. For example, a placement decision of a pod on a node is represented by the table in Figure 4 with decision variables (`node_name`) annotated as `@variable_columns` and other *input variables* supplied by the database.

Constraints Next, the developer specifies constraints against the cluster state as a collection of SQL views. DCM supports both hard and soft constraints, encompassing *all* the cluster management *policies* that the system must enforce.

Hard constraints are specified as SQL views with the annotation `@hard_constraint`. For example, consider the constraint in Figure 5, which states that pod P can be scheduled on node N if N has not been marked unschedulable by the operator, is not under resource pressure, and reports as being ready to accept new pods. We implement this by declaring a view, `constraint_node_predicates`, with a `check` clause

```
-- @variable_columns (node_name)
create table pods_to_assign
(
  pod_name varchar(100) not null primary key,
  status varchar(10) not null,
  namespace varchar(100) not null,
  node_name varchar(100),
  ... -- more columns
);
```

Figure 4: A table describing pods waiting to be scheduled. The `@variable_columns` annotation indicates that the `node_name` column should be treated as a set of decision variables. Other columns are input variables, whose values are supplied by the database.

```
create view valid_nodes as
select node_name from node_info
where unschedulable = false and memory_pressure = false
  and out_of_disk = false and disk_pressure = false
  and pid_pressure = false and network_unavailable = false
  and ready = true;

-- @hard_constraint
create view constraint_node_predicates as
select * from pods_to_assign
check (node_name in (select node_name from valid_nodes));
```

Figure 5: A hard constraint to ensure pods that are pending placement are never assigned to nodes that are marked unschedulable by the operator, are under resource pressure, or do not self-report as being ready to accept new pod requests.

that asserts that all pods must be assigned to nodes from the `valid_nodes` view computed in the database.

Soft constraints are also specified as SQL views with annotation `@soft_constraint` and contain a single record storing an integer value. DCM ensures that the computed solution *maximizes* the sum of all soft constraints. For example, consider CPU utilization load balancing policy across nodes in a cluster (Figure 6). We first write a convenience view (`spare_capacity_per_node`) that computes the spare CPU capacity after pod placement. We then describe a soft constraint view (`constraint_load_balance_cpu`) on the minimum spare capacity in the cluster. This forces DCM to compute solutions that maximize the minimum CPU utilization of nodes, thereby spreading pods across the cluster.

Compiler and runtime The DCM interface for programmers is shown in (Figure 8). The first step is invoking the DCM compiler using the schema and constraints as input. This generates a program (e.g., a Java program – §4.1.2) that pulls the required tables from the database, constructs an optimization model, and solves it using a constraint solver. The generated program is compiled using the relevant toolchain (e.g., `javac` – §4.1.2) and loaded into memory. The compiler returns a `Model` object that wraps the loaded program.

When pods are added to the system, the scheduler updates the relevant tables (like `pods_to_assign`). The scheduler

```
create view spare_capacity_per_node as
select (node.available_cpu_capacity
        - sum(pods_to_assign.cpu_request)) as cpu_spare
from node join pods_to_assign
  on pods_to_assign.node_name = node.name
group by node.name;

-- @soft_constraint
create view constraint_load_balance_cpu as
select min(cpu_spare) from spare_capacity_per_node;
```

Figure 6: A soft constraint that maximizes the minimum spare CPU capacity in the cluster for load balancing.

```
-- @hard_constraint
create view constraint_node_affinity as
select * from pods_to_assign
check (pods_to_assign.has_requested_node_affinity = false
       or pods_to_assign.node_name in
         (select node_name
          from candidate_nodes_for_pods
          where pods_to_assign.pod_name =
                candidate_nodes_for_pods.pod_name));
```

Figure 7: A membership constraint to describe node affinity (the pod must only be assigned to nodes it is affine to, as computed in another view `candidate_nodes_for_pods`)

then invokes `model.solve()` (Figure 8) to find an optimal placement for these pods by assigning values to `pods_to_assign.node_name` according to the specified constraints. The call returns a copy of the `pods_to_assign` table with the `node_name` column reflecting the computed optimal placement. The scheduler then uses this data to issue placement commands for each pod via the Kubernetes scheduling API (the same API used by the default scheduler).

Note that DCM treats the state database as the input to every call to `model.solve()`. It does not (and cannot) assume the cluster configuration changes based on the computed solution, because the caller may choose not to apply the solution, there may be errors during reconfiguration or numerous other external events. This is in sharp contrast to prior art that uses handcrafted solver encodings for specific cluster management tasks, where all the cluster state is duplicated within the solver's memory [40, 53, 57, 92] (§8).

Supporting diverse cluster management tasks and tunable search scopes DCM enables developers to arbitrarily tune the search space of a problem by controlling the data within the input tables and views. For example, consider the the set of pods in the `pods_to_assign` table. For fast incremental initial placement, we can populate the table with a fixed size batch of newly created pods only, and compute placement decisions for the entire batch at a time. For a pod preemption model (a kind of global reconfiguration), we populate the same table with previously placed pods and specify additional constraints that assign a bounded number of pods to a "null node" (representing preemption). Similarly, the scheduler may dynamically sample the subset of nodes to be considered for placement in a given iteration.

Operation	Description
`model = DCM.compile(schema)`	Invoke DCM compiler to synthesize an encoder from the SQL schema and constraints
`model.connect(db)`	Establish JDBC connection to the state DB
`model.solve(timeout)`	Solve constraints and return a set of tables

Figure 8: DCM interface

In this manner, DCM allows developers to easily instantiate models that solve different sub-problems within a cluster manager. We implemented three models in our Kubernetes scheduler: one for fast initial placement, a slower timescale pre-emption model, and an admin-triggered tool for de-scheduling [68] which can be used to recover capacity from highly utilized nodes by terminating pods.

4 DCM Design

As we show in §3, DCM enables programmers to specify cluster management policies using a high-level declarative language familiar to most programmers, and code generate the logic that efficiently computes policy-compliant decisions. However, we have to address several key challenges to realize such a capability.

First, given the amount of expertise that is typically required to handcraft efficient optimization models and encodings, it is non-trivial to bridge the gap between the SQL representation of a problem and the generation of a corresponding encoder that interacts with solvers via their respective low-level APIs. We discuss how the DCM compiler synthesizes efficient encoders in §4.1.

Second, given that programmers need systematic ways to test and debug the policies that they write, we describe how we leverage a common solver capability of finding *unsatisfiable cores* to aid in debugging (§4.2).

4.1 DCM compiler

The DCM compiler generates an encoder that produces optimization models according to the database schema and the constraints specified by the developer.

4.1.1 Syntax and expressiveness

The compiler accepts input SQL tables with variable columns of type integer, boolean, and string (floating point is supported if the backend solver supports it, like Gecode [2]). The compiler supports a subset of the SQL query language for constraint specification, including most commonly used constructs (inner join, anti-join, group by, aggregate queries, sub-queries, correlated sub-queries as seen in Figures 5 and 6, `ARRAY` columns), arithmetic and logical expressions (standard Boolean operators, linear arithmetic, comparisons over integers, and equality checks over strings), standard SQL aggre-

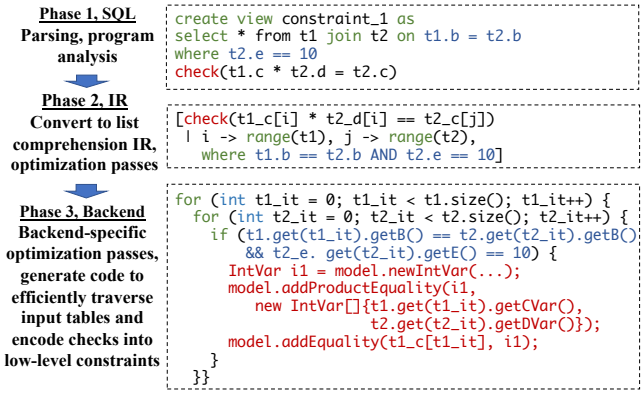


Figure 9: System compiler workflow, showing a simple SQL query, its representation in the IR as a list comprehension, and a simplified version of the corresponding generated code by our or-tools backend.

gate functions (`sum`, `min`, `max`, `count`), and additional aggregates that help in constraint modeling (e.g., the `all_different` aggregate which enforces pair-wise inequalities among a set of variables (§4.1.3)). The compiler is extensible, allowing user-defined aggregates to be added with a few lines of code. We also take special care in dealing with SQL nulls, depending on the backend.

Both SQL tables and views computed in the database can be used as input relations for hard and soft constraints (Figures 5 and 6). This allows developers to efficiently construct inputs for a DCM `model` by exploiting the full extent of the SQL syntax and capabilities supported by the database. For example, the database can efficiently process joins to compute the required inputs for a model.

Using this syntax, we were able to compactly specify all hard and soft constraints encountered in our case studies, including resource capacity, affinity, anti-affinity, and load balancing constraints along various axes. SQL is significantly more concise than the low-level interfaces supported by solvers, with an SQL view compiling down to many low-level constraints.

4.1.2 Compiler workflow

Figure 9 shows the compiler's workflow in generating an encoder from the high-level SQL. The example shown is simplified Java generated by our OR-tools backend.

Phase 1, SQL Internally, the SQL parser first extracts all table and view definitions from the supplied database schema, and produces syntax trees for all the hard and soft constraints. It then performs a series of passes over the queries to infer whether parts of the constraints can be evaluated in the database. For example, 1) simple sub-queries that do not involve variables are better evaluated in the database rather than in the generated encoder or the solver, 2) some backends (like Minizinc) cannot efficiently compute the groups produced by an SQL `group by` because they cannot represent tuples – we can compute these groups in the database as well.

Phase 2, Intermediate representation Next, the compiler converts the query to an intermediate representation (IR) based on list-comprehension syntax [37]. In Figure 9, the SQL query is simplified in the IR as nested for loops and a filtering condition (instead of tables and predicates *across* various clauses of the SQL query). In general, the list comprehension syntax makes it easy to apply standard query optimization algorithms (like unnesting subqueries). It has been well-studied in the database community [23, 24, 37, 45, 55, 56].

Phase 3, Backend The compiler backend generates a program that produces an optimization model by interacting with the interfaces exposed by specific solver toolkits, e.g., setting up linear inequalities for an ILP solver. The IR facilitates support for multiple backends, allowing systems to benefit from different types of solvers. Regardless of the backend, the generated encoders prepare optimization models where variables and constraints are parameterized by the content of the cluster state database. At runtime, the encoder binds these variables to values extracted from the database before dispatching the optimization model to the solver.

OR-tools CP-SAT: Our 'flagship' backend generates encoders for the Google OR-tools CP-SAT solver [3]. It generates Java code to pull cluster state from the database at runtime and iterate over the state to encode constraints efficiently using the CP-SAT solver APIs. It translates joins into hash-table based accesses when feasible (e.g., equality-based joins using primary keys, unique columns specified via the use of SQL `distinct`), or into nested for loops otherwise (Figure 9). We generate several utility classes to aid the encoding (like type-safe tuple classes to refer to records from different tables). The backend performs common sub-expression elimination within the generated code fragments (e.g., when computing a complex expression within `if` or `for` blocks).

MiniZinc: Our initial DCM prototype interacted with the MiniZinc toolkit [80], which exposes a high-level constraint modeling language, and thereby supports integration with a variety of solvers. However, despite our best efforts, we could not scale it to clusters larger than 50-100 nodes due to the limited control we have over Minizinc encodings. However, we use it to interface with tools for debugging models §4.2.

4.1.3 Generating scalable encodings

We now discuss details of our backend for the Google OR-tools CP-SAT solver [3]. A CP-solver encoding specifies different constraints over input variables. For example, to encode a simple intermediate expression $a = (b < 10)$, we need to introduce a constraint $b < 10$, and link the truth value

```
// Unfixed arity
sum([1 | i -> range(t1)
    where t1.c1[i] = 10])
```

```
// Fixed arity
sum([(t1.c1[i] = 10)
     | i -> range(t1)])
```

Figure 10: Two equivalent IR expressions to compute the sum over a subset of a variable column (`t1.c1`).

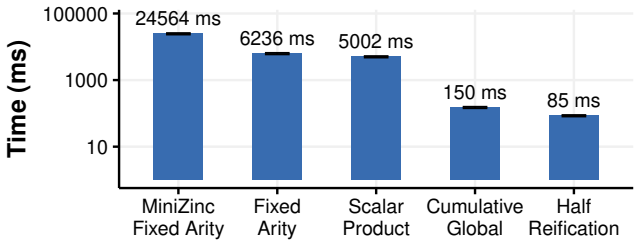


Figure 11: Effect of different optimizations in our or-tools backend on the runtime of a benchmark model (Figure 6). Not shown are option types, where runtimes exceed a minute.

of that constraint to a by introducing two more constraints $a \rightarrow (b < 10)$ and $\neg a \rightarrow (b \geq 10)$ (a process called *full reification*). At a high-level, CP-solvers find feasible solutions via a combination of search and *propagation*: at each node of the search tree, the solver iteratively changes the domain of a variable, causing constraints linked to that variable to update other variables they are linked to, and so on until a fixed point is reached. Therefore, every additional constraint and auxiliary variable we introduce impacts solver performance.

In short, like any constraint solver, the CP-SAT solver's performance is highly sensitive to the encoding (two equivalent encodings often result in vastly different solver runtimes). Hence, we employ various optimizations in DCM compiler based on structural information extracted from the SQL program and IR, to generate encoders that produce efficient models.

The key takeaway is that a literal translation of SQL queries into an encoding is *not scalable*, we therefore have to find smarter encodings. We do so by using re-writing rules that mimic what an optimization expert would otherwise handcraft into an encoder. We describe the impact of these rules using a benchmark that is bottlenecked by the `spare_capacity_per_node` view in Figure 6.

Re-writing to use fixed arity Consider the two expressions in our IR shown in Figure 10. Both these statements correspond to a high-level SQL operation to compute the number of elements in a variable column with values equal to 10. In the first case, we cannot statically determine the size of the filtered list in our encoder, because the values of the variables (and therefore the arity of the list to sum) are not *fixed yet*. The general way to encode such an expression is to use *option types* [75], where a variable might be 'absent'. However, several auxiliary variables and constraints need to be introduced to encode such an expression using option types. This problem is exacerbated by the join in `spare_capacity_per_node`, and where we do not know the arity of the produced *table*. An option type encoding for 1000 nodes and 50 pods for this view produces roughly *200K auxiliary variables and 400K constraints*, and makes our benchmark take more than a minute to complete.

Another approach is to avoid option types, and instead, compute a *sum of predicates* (Figure 10), which achieves the same result because the predicates evaluate to 0 or 1. Doing so keeps the number of auxiliary variables proportional to the number of predicates to evaluate, and brings the benchmark's runtime from minutes to 6.2s (Figure 11). The same encoding in MiniZinc takes 24s to solve, representing the inefficiencies introduced by high-level modeling frameworks.

Re-writing to use scalar products Re-writing to use fixed arity introduces $O(|pods| * |nodes|)$ reified boolean variables and constraints to encode whether a particular combination of rows from both tables appear in the final result set, which is then used to compute the sums required to specify hard constraints (like capacity bounds) and soft constraints (like load balancing requirements). Rather than naively iterate row by row to create several auxiliary variables that are summed to compute the load, our compiler infers that we are effectively computing a scalar product between two vectors, which can be expressed more efficiently with fewer auxiliary variables, which improves runtime by 20% (Figure 11). However, this does not still eliminate the $O(|pods| * |nodes|)$ boolean variables and is still prohibitively slow for large clusters.

Re-writing to use global constraints Global constraints are constraints over *groups* of variables for which solvers implement specialized and efficient propagator algorithms that dramatically reduce the search space of the problem. Encoding a problem using global constraints is key to scaling optimization models to large problem sizes because they typically avoid the need to generate too many auxiliary variables and constraints that burden the solver.

The join discussed above can be further optimized by using global constraints. Observe that we compute the load so that we can specify a capacity constraint on it (the load should be less than a constant). We can therefore detect this possibility and generate code to encode the join and the capacity constraint together as an *interval packing problem*: given a set of interval variables $I_1, I_2 \ldots I_n$ of lengths $S_1, S_2 \ldots S_n$, with demands $D_1, D_2 \ldots D_n$ pack them onto a timeline represented by the intervals, such that the total demand at any given instant of time is less than a capacity C (here, each interval variable represents a cell on the variable column, whereas the timeline represents rows on the column being joined to). This allows us to use a well known global constraint, *cumulative*. Crucially, this non-trivial encoding only introduces $O(|pods| + |nodes|)$ auxiliary variables (instead of $O(|pods| * |nodes|)$), that allows us to scale that SQL query to large cluster sizes, leading to a *96%* reduction in runtime over the previous optimization in our benchmark (Figure 11).

Another example of a global constraint is the `all_different` constraint that ensures a group of variables take different values (a common pattern in various anti-affinity policies). We also leverage membership global constraints where possible to encode the SQL `IN` operator.

As we mention in §4.1.1, we provide a suite of custom aggregate functions that developers may use in their models: these functions take advantage of the above global constraints under the covers.

Full- versus half-reification We already mentioned full reification (e.g., $a \leftrightarrow (b < 10)$). In several cases, it is both correct and more efficient to only introduce *half reified* constraints of the form $a \rightarrow (b < 10)$, because it introduces only half the constraints. Doing so is particularly effective to encode soft constraints for placement preferences in the following form: $b \rightarrow (var = A \vee var = B \ldots)$ (the solver tries to maximize b, which coaxes *var* to draw from a pool of preferred values). Again, using structural information from the IR, we can statically determine when we only need half-reified expressions. In our benchmark, this further yields a 43% improvement in performance (Figure 11).

4.2 Testing and debugging models

An important concern in using DCM is understanding *why* the cluster manager failed in finding a valid solution (e.g., a pod placement decision). Did the cluster run out of resources? Did the user mistakenly specify mutually contradicting constraints, e.g., affinity and anti-affinity over the same group of pods? Or was there a bug in the developer's constraint specification? DCM improves debuggability by taking advantage of a common solver capability: identifying *unsatisfiable cores* [72]. An unsatisfiable core is a minimal subset of model constraints and inputs that suffices to make the overall problem unsatisfiable (e.g., a single input variable that cannot simultaneously satisfy two contradicting constraints).

We leverage this capability by providing a translation layer that extracts an unsatisfiable core from the solver and identifies corresponding SQL constraints and records in the tables that lead to a contradiction. We found this invaluable when debugging complex scenarios involving affinity and anti-affinity constraints. For example, a common pattern we experienced when adding and testing new affinity policies was that the new policy *tightened the problem*, and therefore triggered a violation of some other constraint in the system (such as resource capacity constraints). Rather than suspect a bug in our specification of the new policy, the unsatisfiable core rightfully points us to the contradiction between the capacity constraint and the affinity requirement.

4.3 Implementation

Our DCM implementation is 6.1K LoC in Java for the library and an additional 2.7K LoC for tests. Of this, the OR-tools and MiniZinc backends take up roughly 2K and 1K LOC, respectively. We use the JOOQ library [8] to conveniently interface with different SQL databases. All our experiments use the OR-Tools backend, given that MiniZinc simply does not scale to large cluster sizes. Our implementation is available as an open-source project [17].

5 Experience using DCM

We now describe the three case studies we applied DCM to, and qualitatively assess the development effort to do so. The case studies are presented in the reverse chronological order in which we applied DCM; we found that the overall design and SQL-based programming model were stable throughout the process, even though we did harden DCM along the way. DCM's ability to compute unsatisfiable cores (§4.2) were invaluable in all these efforts.

Kubernetes scheduler This use case is both our most recent and most comprehensive application of DCM. Like the Kubernetes default scheduler, our scheduler also runs as a pod within Kubernetes and uses the same REST APIs and hooks to consume and actuate upon the Kubernetes' cluster state. Our scheduler consumes the same cluster metadata (information about nodes, pods, labels, and other input information) as the default scheduler to make scheduling decisions. Our scheduler uses an embedded in-memory SQL database (H2) as a cache of the cluster state, which is used to serve inputs to `model.solve()` (§3). It computes scheduling decisions for a batch of pods at a time.

Our scheduler is implemented in ~1.5K lines of Java code, with 1.8K lines of code for tests. Roughly half the scheduler's code is boilerplate to subscribe to Kubernetes' state and store it in an in-memory SQL database (H2 [4]). All the tables, views, and policies amount to ~550 lines of SQL. We implemented all policies in Table 2, except H16-20, as they were specific to volume management on GCE, AWS, and Azure.

Of the 550 lines of SQL, roughly two-thirds describe input tables and views that are executed entirely in the database, whereas the rest were used to describe constraints. We were able to handle heavy and complex joins in the database (e.g. computing groups of pods that repel each other due to inter pod anti-affinity). Support for SQL `ARRAY` columns was critical to bounding the size of inputs to DCM.

We spent the vast majority of our effort trying to understand Kubernetes' semantics. Particularly, Kubernetes' *match expression* logic, a DSL used to filter objects based on *labels*, was widely used within several policies supported by the scheduler (taints, tolerations, node/pod (anti) affinities, etc.). Its semantics differed arbitrarily across policies (multiple affinity requirements for nodes were treated as a logical OR, whereas the equivalent for pods was treated as a logical AND [11, 12, 90]). Once we understood the semantics, translating the requirements into SQL was straightforward: it took

a few hours per policy to design and code (i) the schema and constraints, (ii) the logic to write Kubernetes' state into the database, and (iii) the required unit and integration tests.

Most of the time and effort in performance engineering the scheduler went into ensuring that the views executed by the database used the right indexes. While these views could all be expressed concisely in SQL (<20 LOC), the most concise SQL was sometimes not the most performant. For example, an `OR` in some join predicates caused H2 to not use indexes and revert to scans. We had to split these queries into two and `UNION` the results together to meet our performance needs [6]. In another case, we used triggers to simulate materialized views [5], to incrementally update resource reservation counters on each node as pods were placed (rather than compute these statistics each time using a join during pod placement).

VM load balancing utility We built a tool for suggesting VM migrations in a commercial data-center management solution. The system has a resource manager that makes VM placement decisions at various timescales. At slower timescales (e.g., every five minutes), the system introspects the state of the entire cluster and identifies a series of VM migrations to make, with the objective of reducing the overall standard deviation of node resource utilization (along multiple resource dimensions). We apply DCM to improve load balancing in §6.2. The load balancing and capacity constraints we introduced were structurally similar to the ones we specified in our Kubernetes use case.

Distributed transactional datastore We implemented a management plane from scratch for a distributed transactional data platform used in a commercial product. Nodes in the system assume one or more 'roles', such as being a serializer (as in Megastore [20], Omid [25]), a backup serializer, data nodes (that host data shards and replicas), or management nodes. We apply DCM to the management node logic, supporting several requirements provided by engineers. We replicated existing failure handling policies and added new capabilities like distributing roles across nodes, and rack-aware placement of data shards. All policies used <10 lines of SQL, as the system was simple compared to our other use cases.

6 Evaluation

The premise of our work is that DCM is a viable approach to building cluster managers that can (Q1) scale, (Q2) compute high-quality decisions, and (Q3) be easily extended. We answer these questions as follows:

Q1: For scalability: we study the Kubernetes scheduler we built using DCM and evaluate it on a 500 node cluster on Amazon EC2 using workload characteristics from Azure [77]. We use simulations to study scalability up to 10K nodes.

Q2: For decision quality: we study scenarios involving our Kubernetes scheduler as well as the load balancer we built for the commercial virtual machine management platform.

Q3: For extensibility: we were able to express all cluster management policies in our three use cases using SQL. In addition, we discuss a non-trivial extension we built for our Kubernetes scheduler using DCM.

6.1 Q1: Scalability evaluation

We set up a 500 node Kubernetes cluster running on AWS.[1] We use t3.2xlarge instances (8 vCPUs, 32 GB RAM) for the Kubernetes master node and t3.small instances for worker nodes, given that in these experiments, the focus is on the schedulers running on the master node.

Workload We use a publicly available trace from Azure [77], that describes a month's worth of workload information for two million VMs in 2019[2]. It gives us a trace of replicas that were launched, with their corresponding CPU and memory reservations. We replay 14 hours worth of traces and speed them up by 20× to achieve arrival rates seen in Borg clusters at Google [95] (median/peak of 100/500 pod creations per second). We then replay three variants of the workload, each with a fraction **F** of replica groups configured with inter-pod anti affinities within the group; **F** = 100% is a Kubernetes best practice for availability reasons [1], and users even run automated tools like `kube-score` [9] to prevent pods from being deployed without anti-affinities configured. The anti-affinity policy is a challenging constraint because it requires reasoning across groups of pods and uses several handcrafted performance optimizations in the Kubernetes default scheduler (§2). In addition, the workload also exercises most hard and soft constraints shown in Table 2.

End-to-end latency results Figure 12 shows the end-to-end latency for bringing up pods on the AWS cluster, for different values of **F**. The latency is measured from when the workload generator issues a pod creation command to when the pod first changes its status to `Running`. The default scheduler's end-to-end latency degrades as more pods are configured with anti-affinity constraints, with its 95th percentile latency degrading from 4.14s at $F = 0$ to 12.45s at $F = 100$. On the other hand, DCM incurs a higher latency than the default scheduler at $F = 0$ due to the added latency in its critical path from the database and solver (p95 of 5.33s). However, DCM's end-to-end latency characteristics are insensitive to the fraction of pods configured with constraints (ECDFs for DCM are identical across all F, Figure 12). At $F = 100$, DCM improves the 95th percentile end-to-end latency over the default scheduler by 53% (5.9s vs 12.45s).

[1] Kubernetes requires careful configuration and tuning to scale beyond 500 nodes. Even at this size, we had to overcome several issues around networking, kubelet failures, and API throttling to stabilize the cluster [27,70]. We defer to simulations to stress DCM beyond 500 nodes.

[2] Our results are qualitatively similar when using the 2017 trace.

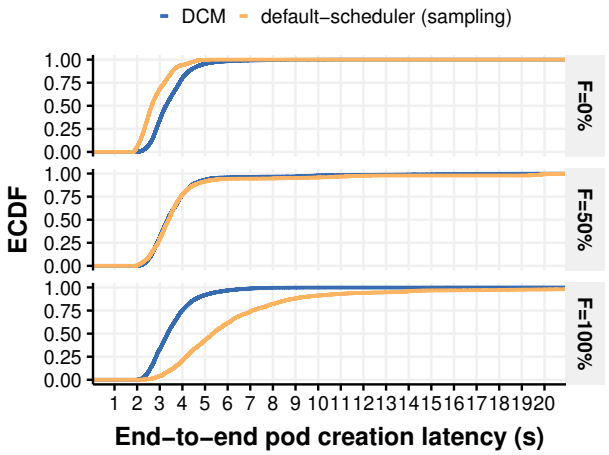


Figure 12: End-to-end pod creation latency on a 500 node AWS cluster, using workload characteristics from the Azure 2019 trace [77] (trace sped up by 20×). **F** is the percentage of pods configured with anti-affinity constraints.

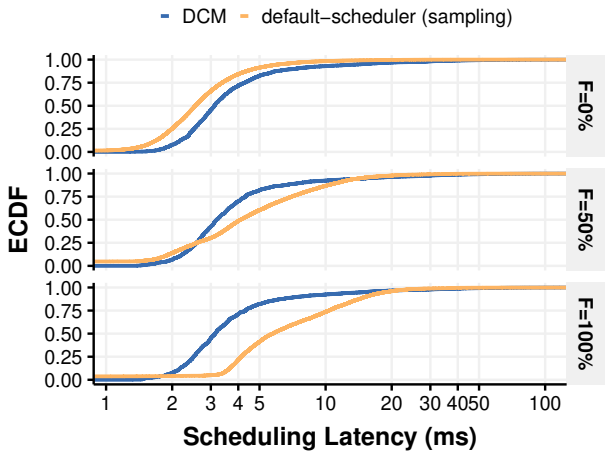


Figure 13: Per-pod scheduling latency at 20× trace speed up (log-scale). Despite our use of a full-featured SQL database and a constraint solver in the critical path, DCM improves per-pod scheduling latency at higher values of F.

Note that our scheduler evaluates all nodes per scheduling decision, whereas the default scheduler only evaluates half the nodes in the cluster for scalability reasons (the number of nodes sampled depends on the total cluster size [93]). When we configured the Kubernetes scheduler to evaluate all nodes, its average latency **doubled** over DCM. Furthermore, the default scheduler incurs a significant amount of engineering complexity in the form of caching and pre-computing optimizations for the sole purpose of speeding up anti-affinity predicates. In contrast, DCM only required a simple SQL specification of the same constraints using 4 SQL views.

Per-pod scheduling latency Figure 13 shows the per-pod scheduling latency for DCM versus the baseline. For DCM, we measure the amortized latency over a batch of pods (maximum batch size of 50 pods), which includes the time taken for querying data from the database, creating a model based on the input data, running the solver, and returning results to the calling code. For Kubernetes, to be conservative, we only measure the time taken to execute all predicates and priorities for a pod once that pod is pulled from the scheduler's work queue. DCM's scheduling latency is competitive with the baseline at $F = 0$: DCM experiences a median (p95) pod scheduling latency of 3.11ms (14.46ms) versus 2.55ms (6.19ms) for the default scheduler. However, DCM's per-pod scheduling latency is similar across all tested values of F, whereas the default scheduler's latency increases significantly with F. At $F = 100$, DCM improves average latency by 1.6× (5.13ms versus 8.04ms). The p95 latency for DCM to schedule a batch of pods stayed under 250ms in all cases. DCM's ability to schedule a batch of pods at a time is what leads to larger absolute savings in end-to-end latency (Figure 12) versus the per-pod scheduling latency.

#Nodes	N=500	N=5K	N=10K
#Variables	5524	45983	91010

Table 1: Average number of model variables before the OR-tools presolve phase.

DCM scheduling latency breakdown Figure 14 breaks down the scheduling latency in DCM by its various phases: the time to fetch inputs from the database (`database`), to encode the inputs into an optimization model (`modelCreation`), and to run the solver (`orToolsTotal`). We also plot the time spent within the or-tools *presolve* phase (`presolve`), where the solver applies several complex optimizations to simplify the supplied encoding. `dcmSolve` subsumes `modelCreation` and `orToolsTotal`. The sum of `dcmSolve` and `database` equals the total scheduling latency.

At a cluster size of 500 and $F = 0$, fetching the required inputs from the database is inexpensive (mean 0.58ms per-pod) compared to invoking the solver (2.8ms per-pod) (Figure 14). DCM's generated code is highly efficient at model creation, contributing an average latency of 450μs per-pod. Similarly, presolve times are also low, staying around 2.5ms for 95% of cases. As we increase F, the database's latency gradually increases (mean 0.88ms) due to the views computed in the database for finding groups of pods that repel each other, but the increase remains small relative to the overall latency. Importantly, solver latency is largely unaffected by the more complex constraints.

Effect of increased cluster sizes To study the impact of cluster size and scale on DCM, we turn to simulations. This is straightforward to do in our scheduler implementation: we simply mock the Kubernetes API, mimicking cluster sizes of 500, 5000, and 10000 nodes. It allows us to replay the same Azure traces against an identical DCM scheduler, but subject the system to a variety of scales and loads. To stress DCM,

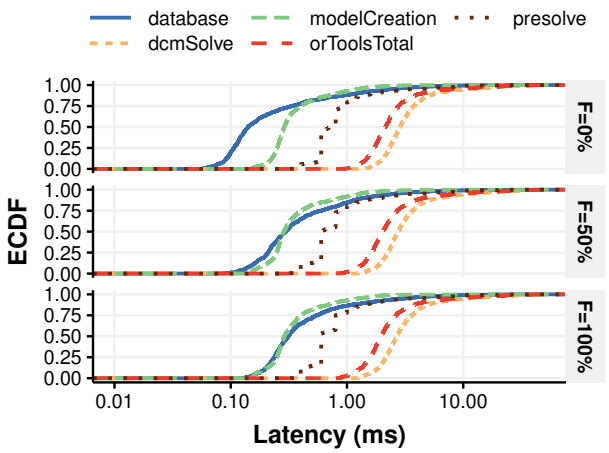


Figure 14: Scheduling latency breakdown between time spent fetching input data from the database (`database`), our generated code creating an optimization model (`modelCreation`), the 'pre-solve' phase in or-tools (`presolve`), and the total solve time in or-tools (`orToolsTotal`).

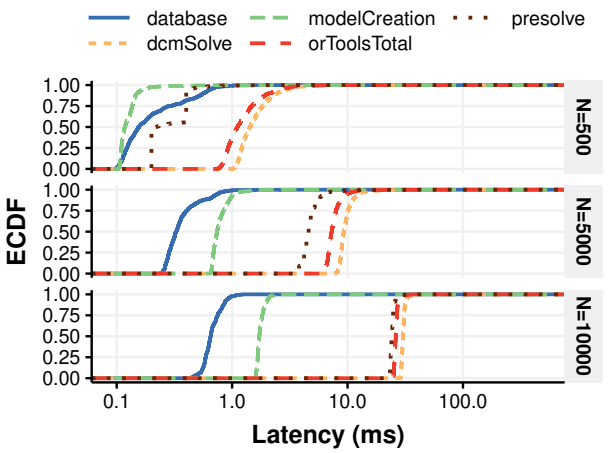


Figure 15: Simulation: scheduling latency breakdown at different cluster sizes. Per-pod scheduling latencies stay under 30ms even at 10K node scale. At 5K and 10K node scales, the presolve phase of the solver dominates.

we speed up the trace by 100× and set $F = 100\%$.

We observe the scheduling latency breakdowns again in Figure 15. At 5K node scale, the per-pod scheduling latency is under 13ms (and 690ms per batch) 99% of the time. At 10K node scale, however, the p99 per-pod scheduling latency is 30ms and 1.6s per batch, which is high. To dig deeper, note that the relative contributions of the database and the constraint solver to the overall latency widen with increasing cluster size. As we mentioned before, our scheduler considers **all** nodes when placing pods, which shifts more of the burden to the solver as cluster sizes increase. We note that the `presolve` phase is the primary contributor to the overall latency. This is because of an API limitation in OR-tools around creating interval variables (§4.1.3).

In particular, there are steps within the solver's presolve pass that we could perform efficiently during model creation, but the OR-tools API is not rich enough to permit. This forces our generated code to construct models with redundant variables (proportional to the number of nodes) that the solver internally tidies up into a more compact encoding (specifically, when encoding our capacity constraints). Table 1 shows the average model sizes generated by our encoder – the presolve phase trims these models down by an order of magnitude. The added cost of repeatedly performing this step on every scheduling decision is acceptable at cluster sizes of up to 5000 nodes ($<$ 1ms at $N = 500$ and $<$ 8ms at $N = 5K$, Figure 15). We are reaching out to the or-tools developers to see if the API can be augmented to avoid this cost.

6.2 Q2: Decision quality evaluation

Kubernetes packing efficiency for higher consolidation We now evaluate a common enterprise data center scenario, where cluster sizes are typically small (under fifty nodes), but consolidation rates need to be kept high. In such scenarios, it is imperative for schedulers to find feasible and dense packings. We use a common pod affinity/anti-affinity pattern seen in production workloads [67], where nginx [13] servers in a web application need to be co-located on the same machine as an in-memory Redis cache [84]. We create 30 such applications, each with 10 pods, with pod CPU and memory requirements following an exponential distribution. We generate 35 such workloads, which leads to a different arrival sequence of resource demands per experiment.

We find that DCM places 100% of pods in 29 out of 35 experiments, and in the worst case, places at least 93% of pods across all runs. In contrast, the baseline scheduler packs all pods only in 3 out of 35 instances. This highlights DCM's effectiveness at placing *groups* of pods. Instead, the baseline myopically places one pod at a time, causing it to make decisions that prevent future pods from being placed. Note, if the pods appear well spaced apart in time, or DCM uses smaller batching sizes, its performance will approach that of the baseline.

Kubernetes placement convergence time for preemptions We now test DCM's effectiveness in making global reconfiguration decisions. We replay a workload used to test Kubernetes' preemption logic [65], that creates 3 sets of pods with different priorities. The resource demands are set to accommodate *only* the highest priority pods on the cluster, and the lower priority pods should either not be placed or be preempted. The default scheduler invokes its preemption logic on a pod-by-pod basis and uses a set of heuristics to determine when to retry pods it could not place (e.g according to a backoff policy, and retrying when nodes report status

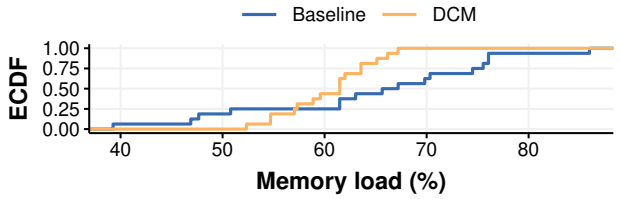


Figure 16: VM load balancing use case: memory load distribution across hosts with and without DCM.

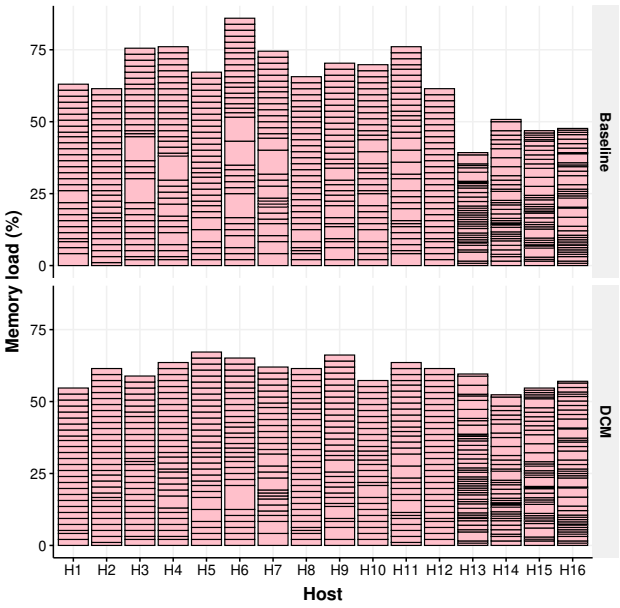


Figure 17: Load distribution before and after using DCM. The X-axis represents hosts, the Y-axis represents the memory utilization per host. Hosts have different capacities, and boxes in each bar represent VM sizes, scaled to the host's capacity.

updates). In doing so, the baseline scheduler takes almost 100 seconds to place all high priority pods. Instead, DCM initially places low priority pods and systematically replaces them with higher priority pods in phases as new pod requests arrive, by invoking its preemption model to look at the state of all nodes (§3). In doing so, the scheduler converges to placing all high priority pods in just under 50 seconds, twice as fast as the baseline scheduler.

VM load balancing quality evaluation We test our VM load balancing tool (§4.3) using a trace from a bug report submitted by a customer. This production cluster has 16 hosts with heterogeneous CPU and memory capacities, and 524 VMs with a range of CPU and memory sizes. The baseline system's heuristic-based load balancer could not identify VM migrations to improve the load distribution of the cluster, which led to the bug report. Figure 16 (baseline) shows the memory utilization of every host as per the trace (we only show memory utilization because there were no CPU resource reservations by the VMs). Figure 17 shows the VM sizes, scaled according to each host's capacity.

With DCM, we specified the necessary hard constraints (capacity and affinity requirements) and a soft constraint that minimizes the load difference between the most and least utilized node. We then asked the tool to identify twenty VM migrations, which significantly improved the load distribution (Figure 16). With the baseline, the most loaded and least loaded nodes were at 85% and 39% utilization, whereas DCM found moves to spread utilization between 52% and 67%. DCM took a second to make its decision, whereas the baseline heuristic takes roughly five seconds.

6.3 Q3: Extensibility

We validate our hypothesis that DCM enables building *extensible* cluster managers. §5 discusses the ease with which we added policies to all three case studies, taking only a few hours per policy to design, implement, and test. In this section, we focus on a more challenging test of extensibility by discussing a non-trivial modification to our Kubernetes scheduler.

Our case study involves a custom Kubernetes distribution where the Kubernetes control plane deploys both pods and VMs (the nodes run both pods and VMs). A challenge here is that the default scheduler's implementation is intricately coupled to the Kubernetes data-structures that represent pods (for example, every predicate and priority implementation expects a pod object). VMs, as a Kubernetes object that can also be placed on nodes, is beyond the Kubernetes scheduler's resource management model. This scheduling inflexibility is a known pain point in the Kubernetes community [66] The only option today is for the Kubernetes scheduler to coordinate with another scheduler that can deploy VMs. This is, therefore, a good case study to validate DCM's extensibility goal.

We extend the Kubernetes scheduler we built using DCM to jointly reason about pods and VMs. From a placement standpoint, pods and VMs are simply *tasks* that need to be assigned to nodes, and only represent a slightly different set of constraints (for example, VMs can be migrated but pods cannot, because most of the Kubernetes ecosystem does not assume pods can be migrated).

Most of our effort went into the Java code and boilerplate required to subscribe to the Kubernetes API to learn about new VM creations (specifically, a Kubernetes Custom Resource [85]) and writing these obtained objects into our database. On the SQL side, however, these capabilities only involved minimal changes to the DCM-based Kubernetes scheduler: we added four constraints in total and made a cosmetic change to the SQL schema for readability (replacing instances of `pods_to_assign` with `tasks_to_assign`). The minimal effort here was possible only because DCM enforces a declarative approach to specifying the cluster state and the constraints on it.

7 Discussion and future work opportunities

Solver Scalability Cluster sizes in enterprises are typically modest and well within Kubernetes' scalability targets. This is a scale that we are confident DCM excels at (§6.1).

Pushing DCM to hyperscale needs, however, is an interesting area for future work. While the techniques described in §4.1.3 are required for scalability, there will inevitably be a point where it is better to *partition a problem*, something that DCM cannot perform automatically for the developer. For example, in a cluster with 100K nodes, it is likely overkill to evaluate every single node in the cluster for an optimal placement decision. Instead, the 100K nodes can be partitioned into 10 or 20 groups that are somewhat similar in composition (analogous to Borg cells [95] or sub-clusters in Hydra [29]). A DCM model (or a custom heuristic) could pick a group to place a workload in, followed by another model that places the workload within the selected group (the second model, in this case, would use as input, a table/view with only nodes from the selected group). Another approach would be to evaluate multiple such groups in parallel and pick the result with the best objective function. We explicitly designed DCM's programming model for such flexibility.

There are several further opportunities for improving performance that we have not yet explored. For example, solvers can be configured to return good-enough (as opposed to optimal) results, when the current best solution is within a certain bound, to improve performance.

Database scalability In-memory, incremental view maintenance is key to scaling the database side. For now, we had to simulate materialized views using triggers in H2 (§5). H2's simplicity also meant that its optimizer did not perform several natural query transformations that more mature engines do, which required us to write more complex SQL than was required (§5). This is additional work that would not be required with an incremental engine. We are currently integrating DCM with the Differential Datalog (ddlog) engine [86].

Expressiveness of SQL So far, across all use cases (§5), we are yet to find a policy we could not express using this model. We are confident of SQL's expressive power for several reasons.

SQL shines at concisely cross-referencing state across different tables, a capability that has been useful in a broad range of contexts (e.g., SQCK [48]). We simply leverage that strength of SQL to both represent complex cluster state and concisely specify constraints spanning several tables; the actual check clauses and objective function expressions within these constraint queries are typically comparable to what is shown in Figures 5, 6, and 7.

The more complex SQL we have written are for views executed in the database, which become inputs for the generated code (§4.1.1, §5). There is a lot of expressive power here; for example, developers may use a database's user-defined functions for specific input transformations if required, but we have not yet needed to do so (even for handling Kubernetes's match expression DSL, §5).

At the same time, DCM does require expressing policies in terms of *intent*, rather than the exact steps of an algorithm. We anticipate that this will pose a learning curve for some developers.

Generality of optimizations DCM cannot prevent users from writing SQL that generates inefficient code, a common challenge for declarative programming models (SQL databases provide tools for users to inspect query plans for this reason, like the `EXPLAIN` query). For now, our compiler warns developers when it emits inefficient code (e.g., cross products across tables without indexes), and exposes detailed diagnostics to understand runtime performance (e.g., Figure 14 and Table 1).

We provide a suite of aggregate functions (like `all_different`) that we encourage developers to use because it leads to clearer policies and makes it straightforward to generate efficient code that uses global constraints (§4.1.1). So far, we only added functions if they were useful across several use cases. It is similar with rewrite rules: we only add ones that have broad utility (e.g., we find the fixed-arity rule applying to most uses of `SUM`/`COUNT`).

8 Related work

Use of solvers for resource management A large body of work has used solvers for resource management, including CP solvers [51] to pack and migrate VMs; flow network solvers [40, 53] and MIPs [38, 42–44, 91, 92] for job scheduling, and ILPs for traffic engineering [30]. These systems use handcrafted encoders written by optimization experts for specific problems. Even in the industry, we find that the few organizations that use solvers for such tasks typically have dedicated teams of optimization experts. In contrast, we generate scalable encoders from a declarative specification written using SQL. Our programming model significantly lowers the barrier to powering systems with constraint solvers – developers express policies directly against the cluster state, without having to translate them into the low-level mathematical formalisms of solver encodings.

We sketched out the initial idea for DCM in a workshop paper [89]. In this paper, we extend this preliminary work with a detailed design and implementation, and a comprehensive evaluation using three case studies.

Quincy [53] and Firmament [40] use flow network solvers for scheduling, which yield quick solve times (sub-second, even for topologies with thousands of nodes), but cannot model many classes of constraints, including inter-task constraints like affinity/anti-affinity [41]. Compared to DCM,

they involve a high degree of modeling complexity: developers need to map scheduling constraints to flow network constructs like vertices, arcs, and flows, which make it challenging to apply to general systems like Kubernetes. For example, the experimental Poseidon Kubernetes scheduler [88] is based on Firmament, but the developers found constraints like inter-pod affinity both hard to implement [87] and scale [16] (up to 3× slower than the default scheduler *evaluating all nodes*).

Wrasse [82] uses a DSL based on a balls and bins abstraction to specify resource allocation constraints and a GPU-based solver to find solutions. Its low-level modeling language makes it hard to express complex constraints; it supports resource capacity constraints, but not other important classes of constraints like affinities and load balancing.

Some production systems use meta-heuristic search for resource management. VMware DRS [47] uses a greedy hill-climbing search, and Service Fabric [76] uses simulated annealing. Several systems employ a variety of heuristics for resource management [26, 28, 29, 32, 33, 39, 54]. These works neither use declarative programming techniques nor benefit from solver-based optimal solutions to enforce policies.

Simplifying systems using relational languages Several works have used the strengths of relational languages to simplify systems programming. Boom Analytics [19] uses the Overlog language to build an HDFS/Hadoop clone with comparable performance. P2 [74] also uses Overlog, but to declaratively specify peer-to-peer overlays. Ravel [96] is an SDN controller that uses SQL databases to abstract and manipulate network state. SQCK [48] simplifies filesystem checker implementations by using declarative queries to validate complex filesystem images instead of writing low-level C code. DCM builds on the above ideas and not only uses a relational database to store and manipulate cluster state but also code generates logic to search for new configurations based on constraints written in SQL.

Network configuration synthesis Network configuration synthesis from high-level specification [22, 34, 35] for BGP and OSPF is orthogonal to dynamic cluster management with constraint specification by DCM. ConfigAssure [79] and Alloy [78] use model finding to identify configurations that satisfy a specification given by an administrator (or detect errors in existing ones). Alloy uses a DSL for specification, whereas ConfigAssure uses the Prolog language. DCM, on the other hand, works on top of standard SQL databases and is capable of supporting optimization goals as well. The tested use cases for ConfigAssure and Alloy are well within scope for DCM.

DSLs for infrastructure automation Many configuration management tools use custom DSLs. Hewson et al. [52] propose an object-oriented DSL to specify a configuration for a data-center, which is enforced by a constraint solver. PoDIM [31] does not use a solver but uses an SQL-like DSL to specify requirements for a configuration. Configuration management tools like Puppet [81], Ansible [83], Terraform [49], and Helm [7] all use custom DSLs to configure and deploy infrastructure repeatably. These systems target a different use case than DCM: they are not designed to solve optimization tasks within a dynamic distributed system at short timescales but instead target infrastructure deployment, which runs at much slower timescales.

9 Conclusion

Cluster management logic is notoriously hard to develop, given that they routinely involve combinatorial optimization tasks that cannot be efficiently solved using best-effort heuristics. With DCM, we propose building cluster managers where the implementation to compute policy-compliant decisions is synthesized by a compiler from a high-level specification. DCM significantly lowers the barrier to building cluster managers that scale, compute high-quality decisions, and are easy to evolve with new features over time. We validate our thesis by applying DCM to three production use cases: we built a Kubernetes scheduler that is faster and more flexible than the heavily optimized default scheduler, improved load balancing quality in a virtual machine management solution, and easily added features to a distributed transactional data store.

Acknowledgements

We thank our shepherd Lidong Zhou and the anonymous reviewers for their valuable feedback. We are grateful to Mihai Budiu, Jon Howell, Sujata Banerjee, and Jacques Chester for their valuable inputs that helped shape this project.

References

[1] 10 most common mistakes using Kubernetes. https://blog.pipetail.io/posts/2020-05-04-most-common-mistakes-k8s/.

[2] Gecode. https://www.gecode.org/.

[3] Google OR-Tools. https://developers.google.com/optimization/.

[4] H2 Database. https://github.com/h2database/h2database/.

[5] H2 Database Features: Triggers. https://h2database.com/html/features.html#triggers.

[6] H2 Database: Performance. https://h2database.com/html/performance.html.

[7] Helm. https://helm.sh/.

[8] JOOQ. https://github.com/jOOQ/jOOQ.

[9] Kube Score. https://github.com/zegl/kube-score.

[10] Kubernetes. http://github.com/kubernetes/kubernetes.

[11] Kubernetes Issue 52141: Multiple matchExpressions in nodeSelectorTerms works unexpectedly. https://github.com/kubernetes/kubernetes/issues/52141.

[12] Kubernetes Issue 70394: s/ORed/ANDed/ nodeSelectorTerms matchExpressions. https://github.com/kubernetes/kubernetes/pull/70394#issuecomment-434127780.

[13] Nginx. http://nginx.org/en/docs/http/load_balancing.html.

[14] Openshift. https://www.openshift.com/.

[15] Openstack. https://www.openstack.org/.

[16] Poseidon benchmarks. https://github.com/kubernetes-sigs/poseidon/blob/master/docs/benchmark/README.md.

[17] Declarative Cluster Management Github Repository. https://github.com/vmware/declarative-cluster-management/, 2019.

[18] Susanne Albers and Michael Mitzenmacher. Average-case analyses of first fit and random fit bin packing. *Random Structures & Algorithms*, 16(3):240–259, 2000.

[19] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 223–236, New York, NY, USA, 2010. Association for Computing Machinery.

[20] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[21] Nikhil Bansal, Alberto Caprara, and Maxim Sviridenko. Improved approximation algorithms for multidimensional bin packing problems. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 697–708. IEEE, 2006.

[22] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 437–451, New York, NY, USA, 2017. ACM.

[23] Kevin Beyer, Don Chambérlin, Latha S. Colby, Fatma Özcan, Hamid Pirahesh, and Yu Xu. Extending XQuery for Analytics. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 503–514, New York, NY, USA, 2005. ACM.

[24] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery, 2002. Retrieved March 2019.

[25] Edward Bortnikov, Eshcar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, Sameer Paranjpye, Francisco Perez-Sorrosal, and Ohad Shacham. Omid, reloaded: Scalable and highly-available transaction processing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 167–180, Santa Clara, CA, 2017. USENIX Association.

[26] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–300, Broomfield, CO, October 2014. USENIX Association.

[27] Architecting Kubernetes clusters. https://learnk8s.io/kubernetes-node-size.

[28] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, SOCC '14, pages 2:1–2:14, New York, NY, USA, 2014. ACM.

[29] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: A federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, Boston, MA, February 2019. USENIX Association.

[30] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *2012 Proceedings IEEE INFOCOM*, pages 846–854. IEEE, 2012.

[31] Thomas Delaet and Wouter Joosen. Podim: A language for high-level configuration management. In *LISA*, volume 7, pages 1–13, 2007.

[32] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[33] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. *SIGARCH Comput. Archit. News*, 42(1):127–144, February 2014.

[34] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 261–281, Cham, 2017. Springer International Publishing.

[35] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, 2018. USENIX Association.

[36] etcd. etcd. https://github.com/coreos/etcd, 2014.

[37] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, December 2000.

[38] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 365–378, New York, NY, USA, 2013. Association for Computing Machinery.

[40] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, Savannah, GA, 2016. USENIX Association.

[41] Ionel Corneliu Gog. Flexible and efficient computation in large data centres, 2018.

[42] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 455–466, New York, NY, USA, 2014. Association for Computing Machinery.

[43] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 65–80, USA, 2016. USENIX Association.

[44] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 81–97, USA, 2016. USENIX Association.

[45] Torsten Grust. Monoid Comprehensions as a Target for the Translation of OQL. In *Workshop on performance enhancement in object bases, Schloss Dagstuhl*, 1996.

[46] B. Guenter, N. Jain, and C. Williams. Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In *2011 Proceedings IEEE INFOCOM*, pages 1332–1340, April 2011.

[47] Ajay Gulati and Xiaoyun Zhu. VMware distributed resource management: design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[48] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 131–146, USA, 2008. USENIX Association.

[49] HashiCorp. Terraform. https://www.terraform.io/, 2014.

[50] HashiCorp. Nomad. https://www.nomadproject.io/docs/internals/scheduling.html, 2015.

[51] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: A consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.

[52] John A Hewson, Paul Anderson, and Andrew D Gordon. A declarative approach to automated configuration. In *LISA*, volume 12, pages 51–66, 2012.

[53] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM Symposium on Operating systems principles (SOSP)*, pages 261–276. ACM, 2009.

[54] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 117–134, USA, 2016. USENIX Association.

[55] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.*, 9(12):972–983, August 2016.

[56] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, number EPFL-CONF-203677, 2015.

[57] Arie MCA Koster, Manuel Kutschka, and Christian Raack. Towards robust network design using integer linear programming techniques. In *Next Generation Internet (NGI), 2010 6th EURO-NF Conference on*, pages 1–8. IEEE, 2010.

[58] Kubernetes. Add a new predicate: max replicas limit per node. https://github.com/kubernetes/kubernetes/pull/71930, 2018.

[59] Kubernetes. Add max number of replicas per node/topologyKey to pod anti-affinity. https://github.com/kubernetes/kubernetes/issues/40358, 2018.

[60] Kubernetes. Affinity/Anti-Affinity Optimization of Pod Being Scheduled #67788. https://github.com/kubernetes/kubernetes/pull/67788, 2018.

[61] Kubernetes. Allow Minimum (or Maximum) Pods per failure zone. https://github.com/kubernetes/kubernetes/issues/66533, 2018.

[62] Kubernetes. Maximum of N per topology value. https://github.com/kubernetes/kubernetes/pull/41718, 2018.

[63] Kubernetes. MaxPodsPerNode - be able to set hard and soft limits for deployments / replicasets. https://github.com/kubernetes/kubernetes/issues/63560, 2018.

[64] Kubernetes. Pod priorities and preemption. https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/, 2018.

[65] Kubernetes. Scheduler sometimes preempts unnecessary pods. https://github.com/kubernetes/kubernetes/issues/70622, 2018.

[66] Kubernetes. Add custom resource scheduling. https://github.com/kubernetes/kubernetes/issues/82118, 2019.

[67] Kubernetes. Assigning Pods to Nodes. https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#more-practical-use-cases, 2019.

[68] Kubernetes. Kubernetes Descheduler. https://github.com/kubernetes-sigs/descheduler, 2020.

[69] Kubernetes mailing list. Let's remove ServiceAffinity . https://groups.google.com/forum/#!topic/kubernetes-sig-scheduling/ewz4TYJgL0M, 2018.

[70] Kubernetes Master Tier For 1000 Nodes Scale. https://tinyurl.com/y97ysbrd.

[71] KubeVirt. https://kubevirt.io/.

[72] Kevin Leo and Guido Tack. Debugging unsatisfiable constraint models. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming*, pages 77–93, Cham, 2017. Springer International Publishing.

[73] Kubernetes Topology Manager Limitations. https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/#known-limitations.

[74] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 75–90, New York, NY, USA, 2005. Association for Computing Machinery.

[75] Christopher Mears, Andreas Schutt, Peter J. Stuckey, Guido Tack, Kim Marriott, and Mark Wallace. Modelling with option types in MiniZinc. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming*, pages 88–103, Cham, 2014. Springer International Publishing.

[76] Microsoft. Service Fabric. https://tinyurl.com/y728dctp, 2016.

[77] Microsoft. Azure Public Dataset. https://github.com/Azure/AzurePublicDataset, 2017.

[78] Sanjai Narain et al. Network configuration management via model finding. In *LISA*, volume 5, pages 15–15, 2005.

[79] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage.*, 16:235–258, 09 2008.

[80] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[81] Puppet Labs. Puppet. https://puppet.com/, 2005.

[82] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 15:1–15:12, New York, NY, USA, 2012. ACM.

[83] Red Hat. Ansible. https://www.ansible.com/, 2012.

[84] Redis. Redis. http://redis.io/, 2009.

[85] Kubernetes Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[86] Leonid Ryzhyk and Mihai Budiu. Differential datalog. In *Datalog 2.0*, Philadelphia, PA, June 4-5 2019.

[87] SIG Scheduling. Affinity/Anti-Affinity Update . https://groups.google.com/forum/#!msg/kubernetes-sig-scheduling/nHWb9zCMOyo/tkbtFf8lBgAJ, 2018.

[88] SIG Scheduling. Poseidon . http://github.com/kubernetes-sigs/poseidon, 2018.

[89] Lalith Suresh, João Loff, Nina Narodytska, Leonid Ryzhyk, Mooly Sagiv, and Brian Oki. Synthesizing cluster management code for distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 45–50, New York, NY, USA, 2019. Association for Computing Machinery.

[90] Assigning Pods to Nodes: affinity and anti affinity. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity.

[91] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[92] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.

[93] Kubernetes Scheduler Performance Tuning. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning/.

[94] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[95] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 18:1–18:17, Bordeaux, France, 2015. ACM.

[96] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A database-defined network. In *Proceedings of the Symposium on SDN Research*, SOSR '16, New York, NY, USA, 2016. Association for Computing Machinery.

Protean: VM Allocation Service at Scale

Ori Hadary Luke Marshall Ishai Menache Abhisek Pan
Esaias E Greeff David Dion Star Dorminey Shailesh Joshi
Yang Chen Mark Russinovich Thomas Moscibroda *

Microsoft Azure and Microsoft Research

Abstract

We describe the design and implementation of Protean – the Microsoft Azure service responsible for allocating Virtual Machines (VMs) to millions of servers around the globe. A single instance of Protean serves an entire availability zone (10-100k machines), facilitating seamless failover and scale-out to customers. The design has proven robust, enabling a substantial expansion of VM offerings and features with minimal changes to the core infrastructure. In particular, Protean preserves a clear separation between *policy* and *mechanisms*. From a policy perspective, a flexible rule-based Allocation Agent (AA) allows Protean to efficiently address multiple constraints and performance criteria, and adapt to different conditions. On the system side, a multi-layer caching mechanism expedites the allocation process, achieving turnaround times of few milliseconds. A slight compromise on allocation quality enables multiple AAs to run concurrently on the same inventory, resulting in increased throughput with negligible conflict rate. Our results from both simulations and production demonstrate that Protean achieves high throughput and utilization (85-90% on a key utilization metric), while satisfying user-specific requirements. We also demonstrate how Protean is adapted to handle capacity crunch conditions, by zooming in on spikes caused by COVID-19.

1 Introduction

The Cloud has revolutionized the way computing resources are consumed. Providers allow end-users easy access to secure, elastic and state-of-the-art resources, while applying efficient management techniques in order to optimize their return on investment. In particular, resource *virtualization* is used to maximize the utilization of the underlying hardware. Consequently, one of the most crucial components in the cloud stack is the Virtual Machine (VM) *allocator*, which assigns VM requests to the physical hardware. Indeed, sub-optimal placement decisions can result in fragmentation (and in turn, unnecessary over-provisioning of physical resources), performance impact and service delays, and even rejection of incoming requests and customer impacting allocation failures.

In this paper, we describe in detail the VM allocator for Azure – one of the leading cloud service providers in the world. Azure provides and manages infrastructure for SAAS, PAAS and IAAS workloads. Its fleet consists of millions of physical machines spanning more than a hundred countries. Azure offers more than 500 different VM types tailored to a vast array of application requirements reflected in the virtual resource specification of each VM. VMs serve as the primary units of (multi-dimensional) resource allocation, and the means through which customers are able to leverage the rich array of computing services offered by Azure; see §2 for an analysis of Azure workloads.

The rapid growth of Azure both in terms of its feature set and massive geo-scale mandated that its core VM allocator be designed in a *robust* manner. First, the allocator logic must be *extensible* – to efficiently facilitate new features, constraints and offerings over time. Second, close attention was given to *flexibility* – in our context, the ability to configure the allocator to different working conditions and scenarios. Third, the core algorithms had to be highly *optimized*: Given Azure's scale, even 1% in fragmentation reduction can lead to cost savings in the order of $100M per year.

From an operational perspective, the total demand in Azure is in the order of millions of VMs per day. Such large scale leads to a complicated system challenge – satisfying this high request rate while maintaining fast response times and high resource utilization. In principle, an allocation service needs to control a sufficiently large inventory of underlying capacity (or domain), so that new requests assigned to the domain can be accommodated, and customers within the domain can scale-out (namely, get additional VMs upon request). However, controlling a large inventory inherently impacts the *latency* of an allocation. To avoid unacceptable delays, a design must include efficient mechanisms for determining the physical placement of the VM. In addition, to achieve adequate *throughput*, the system architecture may incorporate multiple allocation processes [43]. Parallelizing the allocation logic introduces new challenges, such as sustaining high resource utilization while keeping conflicts to a minimum.

While some of these challenges have been discussed in similar contexts [17, 38, 43, 48], most previous works either do not provide full details on the design and implementation, or resort to simulation studies (or small size implementations) without providing comprehensive evaluation from a global-scale production deployment. In this paper, we describe the

*O.H, L.M, I.M and A.P contributed equally to this paper.

design, implementation and evaluation of Protean, the core allocation service governing all VM placement and resource allocation in Azure. An instance of Protean operates at the granularity of an availability zone (typically 10-100 thousands of machines), which allows for high acceptance rates and seamless scale-out capabilities.

To achieve the desired robustness while controlling such large inventories, Protean's design provides a clear separation between *policy* and system *mechanisms*. Policy is expressed through a flexible rule-based Allocation Agent (AA), which addresses multiple constraints and performance criteria for allocating VMs. AA's logic is inherently *extensible*; rules can be refined and added with minimal disruption to the system. Crucially, the rule-based semantics foster *explainability*, and force clear and conscious trade-offs between the numerous metrics and optimization criteria. On the system side, a multi-layer caching infrastructure keeps track of previous allocation outcomes through efficient update mechanisms, resulting in an order of magnitude reduction in turnaround time compared to a system without this caching layer. Notably, the memory footprint of the cache is manageable (e.g., around 1GB for 10k machines), and scales sublinearly with the number of machines. By slightly compromising on the allocation quality, we enable multiple AAs to run concurrently, resulting in increased throughput with negligible conflict rate. The number of AAs, as well as key rule parameters, are tuned at a slow time-scale using production data.

Our results from real production measurements and a variety of simulations demonstrate that Protean achieves low latency (typically 20ms per VM), while satisfying user-specific requirements and values of 85-90% for a key utilization metric. Importantly, Protean can easily satisfy the peak demands observed in production (up to 2000 requests per second), and may sustain much higher throughput if needed, as demonstrated in simulations §6.2. In addition, we show how Protean adapts to different conditions, by focusing on recent capacity challenges during the COVID-19 crisis. In particular, we discuss how Protean seamlessly allowed critical control-plane policy changes that were required to support the sudden increase in demand. In summary, our main contributions are:

- We provide a detailed analysis of the workload and inventory of Azure. Our analysis (§2) reveals key characteristics, which motivate Protean's design.
- We design a flexible rule-based allocation agent (§3), which allows operators to incorporate new logic and *explain* allocation outcomes to customers.
- To our knowledge, we provide the first detailed account of the allocation logic and key implementation details of a core VM allocator in a leading public cloud provider. Our implementation includes a novel caching infrastructure tailored to expedite the VM allocation process (§5).
- We evaluate Protean using extensive measurements from geo-scale production, and augment these evaluations with low and high fidelity simulators where necessary (§6). Our results show that Protean achieves low latency and high throughput while sustaining high utilization under diverse operating conditions.

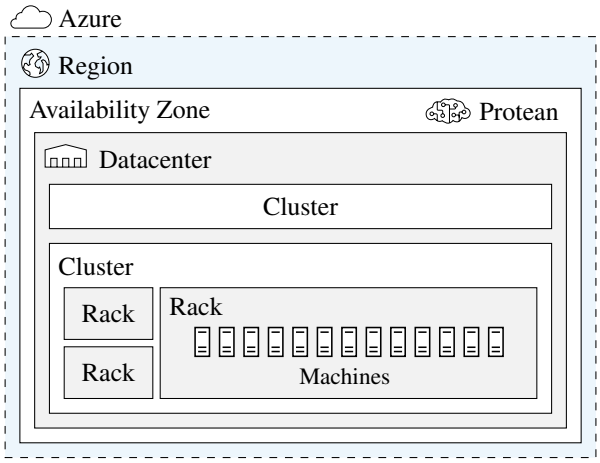


Figure 1: Azure Cloud Topology: Regions consist of availability zones, each of which comprise several datacenters that house racks of servers.

2 Background and Motivation

2.1 Azure – A Global-Scale Public Cloud

The inventory. The global Azure inventory is arranged in a hierarchy of *regions* and availability *zones*, exposed directly to the customer. A region can have up to three zones, each in turn consisting of one or more *datacenters* (see Fig. 1). Each datacenter is divided into *clusters* and *racks*. There is no strict upper bound on the size of a zone or a datacenter. Our larger zones have over a hundred thousand machines, spread over more than a hundred clusters, with each cluster having around a thousand machines. The smaller zones have only around a thousand machines.

The inventory within a zone is typically heterogeneous, with machines ranging across multiple hardware generations and Stock Keeping Unit (SKU) configurations, including special servers for HPC, GPUs, etc. Table 1 summarizes the distribution of the different hardware generations for one of the zones. In this case, the bulk of the inventory belongs to two generations, while the others represent a generation that is being decommissioned, and a new generation that is in early-stage deployment. In contrast to zone heterogeneity, a cluster is a *homogeneous* set of machines (e.g., identical SKUs and configurations) spanning multiple racks; each cluster supports most VM types. In §3.2 we discuss how we exploit cluster homogeneity to improve request latency.

The workload. As mentioned, Azure exposes numerous options for renting VMs. Users specify their requirements in

Gen	Cores Per Machine	# Machines
3	10	7295
4	24	34208
5	40	18016
6	48	2064

Table 1: Distribution of hardware generations for a zone. Individual similar SKUs are aggregated within each generation.

the form of an *service request*. Each zone may accommodate millions of requests per day. A service request consists of one or more *VM requests*, grouped as a *tenant*. The tenant service model expresses the relationships and constraints imposed on these VMs. An allocation succeeds only if all the requested VMs within a tenant are successfully allocated (gang-scheduling). A service model specifies the type of each VM (which in turn determines the core, memory, disk, or network requirements for the VM), fault domain requirements, and priority of the service. By default, the tenant VMs can be spread across the entire zone. However, a tenant can request all its VMs to be co-located within specific inventory boundaries such as a datacenter or a row, or conversely, can forbid too many of its VMs from being placed on the same machine or rack. A tenant can even specify that no VMs from any other tenant be placed on the machine that hosts its VMs.

A customer can resize (scale in/out) or delete an existing tenant, or create a new tenant. Platform initiated requests due to unexpected machine failures, planned maintenance, or decommissioning of machines can lead to reallocation of some or all tenant VMs. Note that higher-level services can stitch together multiple tenants to expose alternative grouping semantics to customers, such as jobs with tasks that can be incrementally scheduled, or auto-scaled group of identical VMs (e.g., virtual machine scale sets [4]). These services are responsible for breaking the groups of VMs into tenants before sending service requests to Protean.

Protean – a zone allocation service. Azure operates an allocation service for each zone, termed Protean.[1] Requests are assigned to each zone either directly by the customer, or by a higher-level service. The main role of Protean is to find a physical placement (machine) for each VM in an allocation request, subject to explicit requirements and constraints specified in the underlying service model, as well as other internal operational considerations. To cope with large request loads, Protean employs multiple *Allocation Agents (AAs)*, which run in parallel. Similar to [43], each agent is aware of the entire inventory and can choose any eligible machine from the inventory to host a VM. The authoritative state of the inventory is maintained in a persistent store. Each AA maintains its own view of the inventory, which is updated periodically and in response to allocation or inventory related events.

[1] Protean means able to change frequently, versatile.

2.2 Workload Analysis

We next analyze some properties of our workload, with a focus on characteristics that have influenced Protean's design.

Demand is heterogeneous. Our zones exhibit workloads which are fairly diverse in nature. We observe a large number of different VM types, see Table 2. The distribution is generally nonuniform – some VM types may account for up to 50% of the workload, while others are rare. To give more insight into the challenge pertaining to packing the VMs, Table 3 shows the distribution of CPU requirements, measured in number of cores. We observe that most VMs require a small number of cores, but some require half or even an entire server.

VM Type	Zone1 (%)	Zone2 (%)
A	4.6	0.1
B	3.5	3.6
C	6.5	12.9
D	0.7	8.4
E	1.9	3.7
F	3.2	4.4
G	0.6	3.1
H	0.8	2.2
I	2.4	7.4
J	23.7	31.6
K	21.3	2.1
L	3.5	0.4
M	0.0	2.2

Table 2: Distribution of VM types for selected zones. VM types having < 2% in both zones are excluded. We avoid using real VM type names to preserve confidentiality.

Subsequent requests are similar. While our system supports many VM types, we observe that subsequent requests are fairly "similar". For example, Fig. 2 shows the *reuse distance*, which for each request of VM type v, measures the number of unique VM types requested since the last time that v was requested. We observe that more than 80% of requests have zero reuse distance, while the majority has distance less than five. This behavior can be attributed to a combination of factors, such as large service requests that ask for the same type of VM, and having a relatively small set of popular VM

VM Cores	Zone1 (%)	Zone2 (%)
1	17.1	27.0
2	37.4	52.4
4	32.0	10.5
8	8.9	4.5
$\geqslant 10$	4.3	2.6
$\geqslant 20$	0.3	3.1

Table 3: Distribution of VM resources for selected zones.

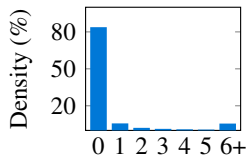


Figure 2: Reuse distance for VM requests in a zone for an entire day.

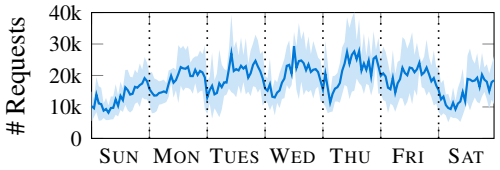


Figure 3: Number of VM requests for a zone. Counted over each hour, and averaged over day-of-week for five weeks (shaded area is standard deviation).

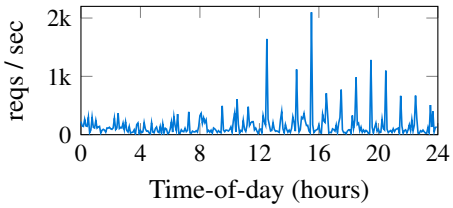


Figure 4: Requests per second for a zone on a single day, with max over 5 min intervals.

types (see Table 2). This "locality" property plays a key role in our design for the allocation agent.

VM lifetime varies substantially. We observe that most VM lifetimes are short, in the order of several minutes. However, some VMs can stay in the system "forever" – for weeks and months. See Figure 5 for empirical lifetime distributions across representative zones.

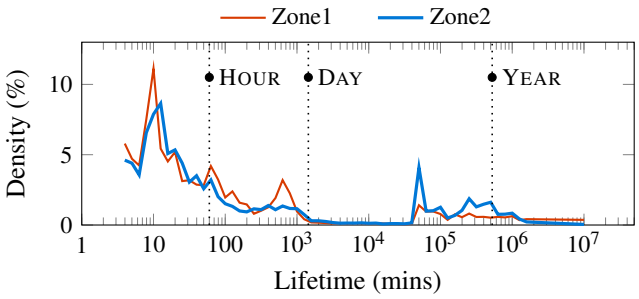


Figure 5: Zone-based VM lifetime distributions. The data represents VMs that were alive during a ten-day period in March 2020. The lifetime of VMs that remained alive when the data was collected (June 2020) may be longer than reported.

Demand has spikes and diurnal pattern. Fig. 3 shows the request count over a week in one of our busier zones, averaged over day-of-week for 5 weeks. We observe some diurnal patterns, e.g., typically less usage overnight. Demand can also have large spikes throughout the day, as seen in Fig. 4. Note that the demand reaches above 2k requests per second. This behavior forces us to provision for the peak by employing multiple AAs for each Protean instance (see §4). At the same time, we exploit the off-peak periods to better prepare the AAs for future allocations (see §5.2).

Tenants sizes are typically small, but can be huge. Our analysis on two large zones indicates that 94% of requests are for a single VM. 99% of requests are for five or less VMs. We also observe a few requests for hundreds of VMs; naturally, such requests would pose additional challenges (e.g., spreading the VMs across different fault domains).

2.3 Takeaways

Scale and uncertainty. Our analysis demonstrates that the incoming demand is highly variable. Because Protean latency and throughput requirements cannot be compromised, our design has to account for extreme demand conditions. Furthermore, Protean has to accommodate small and large regions, which requires flexible configurations (for example, the number of AAs).

Opportunity for caching. We have provided evidence that subsequent requests are similar over time. This motivates the "caching" of placement evaluation logic, and reuse across multiple requests – this idea is central in our design and facilitates scaling to large zones and regions.

The packing challenge. Our workload is highly diverse – numerous VM types of different sizes, high variability in lifetimes (which are unknown in advance). This poses a substantial challenge in adequately "packing" the VM in physical servers. Algorithmically, a simplified version of our packing problem already maps to dynamic bin packing [11], which is an NP-hard problem in the offline setting (i.e., assuming all VM arrivals are known), with quite bad competitive ratio in the online setting [6]. In our practical setting, we have other elements that make the problem even more challenging (multiple priorities, fault domain requirements, etc.). Supplementary to this paper, we release a new trace that can be used by the research community to design and test different packing algorithms [3].

3 Rule-Based Allocation Agent

In this section, we describe the main design principles of Protean's allocation agent.

3.1 Metrics and Constraints

Metrics. Protean targets several metrics related to both performance and quality of the allocation. The key metrics are:

- *Latency.* A single VM allocation should be satisfied promptly, typically within 20 ms.
- *Throughput.* Protean should be able to handle peak demands without delaying or throttling requests.

- *Acceptance.* Protean should minimize the rejection rate. A rejection occurs when a VM request cannot be satisfied.

We note that latency is not important in isolation (although might become excessive for large tenants). Nonetheless, lower latency facilitates higher throughput. Intuitively, when latency is lower, requests can be processed by fewer AAs, resulting in fewer conflicts and likely higher throughput. Furthermore, lower latency decreases the probability that an individual AA drifts from the true state of the inventory, which improves allocation quality. Naturally, the three metrics above depend on the size of the inventory. Latency and throughput become more challenging with larger inventories, but accepting requests becomes easier. We are also interested in efficient usage of compute resources; we will formally define utilization-related metrics in §6.

Requirements and constraints.
Addressing multiple considerations. First and foremost, Protean must make *correct* assignments; for example, an allocation cannot violate the capacity of a machine. Additionally, the request may include certain constraints that the allocation must satisfy. For example, certain VMs may require a specific type of hardware (e.g., GPUs). Furthermore, a service request for multiple VMs may require that the VMs are spread across multiple fault domains (typically across different racks).
Tenant experience. Protean should avoid allocations on machines that are not in a "ready" state; have not been updated with the latest host environment; or are likely to fail in the near future. If a machine fails, then its hosted VMs *must* be allocated to other machines. Low priority VMs are used by Azure offerings, such as Batch [1] and Spot Virtual Machines [2]. While these VMs are allowed to be preempted, Protean still aims to minimize their eviction rates.
Adaptability. Protean must allow for an easy configuration of allocation logic, and adjust for different conditions.
Extensibility and interpretability. Protean should be easily and safely extendable, in order to enable engineers to incorporate new allocation logic. Accordingly, the allocation logic should allow for incremental changes, and performing A/B testing in production. Moreover, Protean should enable operators to interpret the allocation decisions (e.g., "why did the request fail?", "why was machine x chosen for VM request v?"); explaining allocation outcomes is regarded as one of the main challenges in large-scale cloud scheduling [46].

3.2 Allocation Rules

As discussed above, Protean has to account for multiple considerations simultaneously. First, strict placement constraints need to be enforced (e.g., a VM has to be allocated to a specific hardware type); other placement considerations can be viewed as "preferred", for example, it is better to place the VM on a server that is perceived as healthy, has certain disk configuration, etc. On top of that, Protean targets "high-quality" allocations; for example, packing the servers efficiently by minimizing fragmentation, balancing allocations across racks, avoiding lower-priority VMs evictions, etc. Due to the numerous dimensions involved, Protean's allocation logic is organized as a set of *rules*. The rules determine which machine will be assigned for each individual VM. A service request for k VMs will invoke the rule logic k times.

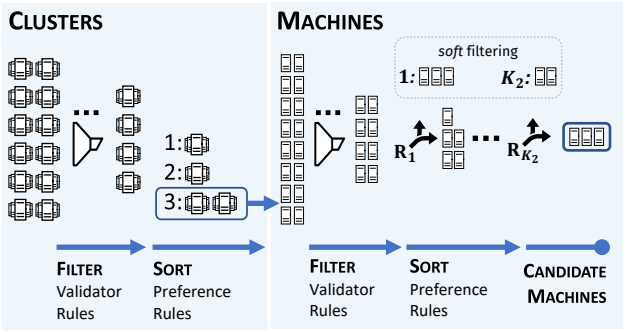


Figure 6: Rule based selection.

Rules are classified into either *validator* or *preference* rules. A validator rule accounts for hard constraints, whereas a preference rule can be viewed as a soft constraint. The rules are arranged in a two-level hierarchy – cluster selection rules followed by machine selection rules (Fig. 6). In total, there are currently around one hundred rules.

Cluster and machine selection rules. Cluster selection rules effectively reduce the time complexity of the selection process by limiting the scope of the machine selection rules to a small number of clusters in a zone. Because clusters are homogeneous (see §2), we implement several cluster validator rules which filter out clusters that are not relevant for the VM request (e.g., a VM that requires a GPU machine can by hosted only on a cluster with GPU machines). In addition, a few cluster preference rules are used to sort the valid clusters (e.g., we prefer emptier clusters to balance the available capacity across clusters). Based on that, Protean chooses the k highest-quality clusters, where $k \in [8, 16]$ is a configurable parameter; the inventory for the machine selection rules will in turn consist of the machines in these clusters. The parameter k is set based on a tradeoff between exposing a large set of machines for making high-quality decisions and sustaining adequate latency.

In turn, machine selection rules again consist of validator rules that exclude specific machines from being considered, followed by preference rules which eventually select a small number of machines that are the best match for the particular VM. A randomized tie-breaking rule picks one of these machines for the physical assignment of the VM. In what follows we provide a more formal description of rule semantics, as well as some examples.

Validator rules. Each validator rule implements the Boolean method IsValid(x, v) to indicate whether an object x is a valid candidate for placing VM v; an object can be either a cluster or

a machine, depending on the rule. Validator rules are used to prune the set of objects in the inventory to a subset of objects that are valid candidates for placing the VM. **Examples**: (1) AreNodeResourcesValid(x, v) checks whether machine x has enough available capacity to accommodate VM v. The method returns *true* if all resource dimensions can be fulfilled (CPU, memory, disk, etc.). (2) IsTypeSupported(x, v) checks whether cluster x is compatible with the VM type corresponding to v.

Preference rules. A preference rule quantifies the extent to which each candidate object x (either a cluster or a machine, depending on the rule) is a good fit for the VM. Each preference rule r accounts for a specific consideration (e.g., packing quality, balancing, cluster/machine quality, etc.) through a numeric *score* $S_r(x,v)$. We use the convention that a lower score is better. **Examples:** (1) BestFit(x,v) assigns a score $\sum_i w_i(a_i(x) - d_i(v))$, where $a_i(x)$ is the availability of "resource" i (e.g., CPU cores, memory, SSD)[2] in machine x, $d_i(v)$ is the requirement of the VM for that resource, and w_i is the weight of the resource which quantifies its scarcity (intuitively, the higher w_i the scarcer the resource). A lower score here implies that the machine is a better fit for that VM, since it is closer to being fully packed; we note that similar packing heuristics have been proposed in [39]. (2) PreferNonEmptyMachines(x,v) This rule prefers to use machines that are non empty, primarily in order to improve packing quality. (3) PreferEmptierClusters(x,v) This rule quantifies how many empty cores cluster x has. The idea here is to balance the available capacity among clusters. This is done to minimize the probability that the cluster capacity is exhausted, which is important from several perspectives. For example, some customers require affinity within the cluster, and would not be able to scale out if the cluster is completely full.

3.3 Accounting for Multiple Rules

As illustrated in Fig. 6, the sequence of validator rules filters out objects (clusters, machines) that are not eligible for the particular VM. One of the main challenges in the design of Protean was: how to account for multiple preference rules? The inherent issue here is that different rules represent different and hard-to-compare preferences. We describe below the principles of our approach.

Compare method. Each preference rule r implements the Compare$(x, y \mid v)$ method to compare two objects x and y based on their scores; the method returns 0 if scores are equal, 1 if $S_r(x,v) < S_r(y,v)$, or -1 otherwise.

Comparisons and sorting. Each preference rule expresses its relative importance using a *weight* (or gain value). Two objects are compared according to an aggregate score computed as the sum of products of the compare value returned by each rule compare method and its weight. Using the pairwise comparisons, Protean computes a sorted list of the entire set of objects based on their aggregate preference scores.

Weight assignment. While our system allows to set any positive value for the rule weights, we have chosen to set the weights in a way that imposes strict ordering between the preference rules. The rules are assigned weights according to an order-preserving encoding (i.e., weights are exponentially apart from each other), such that, effectively, any rule can only express a preference among objects that have been preferred by the previous rule. Accordingly, the entire set of rules (including validator rules) can be regarded as a *filtering* process in which the set of preferred objects is narrowed as more rules are considered; see §3.4 for discussion.

Quantization. Having a strict prioritization among the preference rules, requires "smoothing" the preference rules, so that all rules can contribute. We do so by quantizing the score of some rules into a small number of buckets (e.g., rules with a continuous score, such as BestFit). For example, we may apply the transformation $\lceil S \cdot N \rceil$, where $S \in [0,1]$ is the original (continuous) score and N is the number of buckets. The rule ordering and the specific quantization values entail domain knowledge and understanding of business needs and preferences. Their setting is based on trial and error, building on simulation results as well as production telemetry.

3.4 Discussion

We conclude this section by discussing how the rule-based allocator helps us achieve our design goals.

Addressing multiple considerations. Having multiple rules allows us to address multiple hard constraints, and explicitly influence the quality of the allocation through designated rules (e.g., best-fit for packing). The fundamental requirement of making "correct" allocations will manifest itself in certain system mechanisms; for example, ensuring that decisions are made based on the true state of the zone (§5).

User experience. By design, Protean will *not* fail a VM request if there exists a feasible assignment for that VM. This holds because (i) Protean chooses clusters that contain feasible nodes; (ii) Protean considers all nodes in the selected clusters. In addition, Protean has rules that target better user experience (e.g., prefer "healthier" machines).

Adaptability. The rules themselves can be customized and refined as needed. For example, if a specific rule is too "aggressive", a simple configuration change can make it softer, e.g., by making the quantization coarser. As a concrete example, we describe in §6 how Protean has been adapted to tackle a capacity crunch during the Covid-19 crisis.

Extensibility and robustness. Our rule-based allocator is inherently extensible and robust. It is not too difficult to insert a new rule, or to modify or delete an existing one; two main design choices enable that: (i) rather than using a general

[2] Protean currently does not account for power. Power budgets are defined for different aggregations of servers: chassis, racks, rows, etc. A separate power-capping system [31] ensures that power usage does not exceed the budget; since power consumption falls within the budget at high percentiles, capping engages rarely.

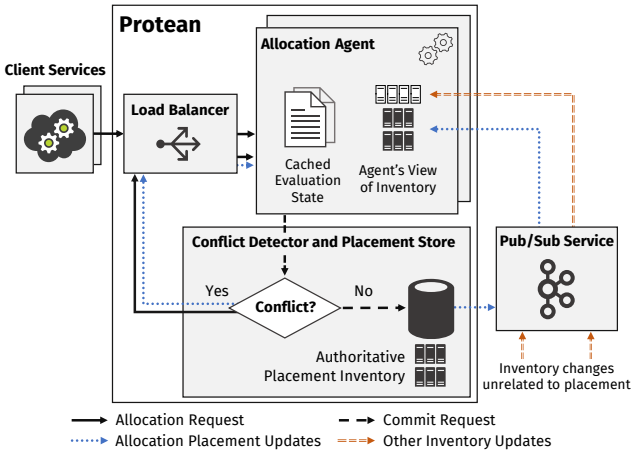


Figure 7: Protean System Architecture

weighted score function, Protean maintains a ranking of the rules, and auto-generates weights that maintain the exponential distance property; (ii) the internal rule scores are not factored in the sorting of the machines, which makes the design or modification of rules more robust. This is enabled by using the rule compare method as a building block.

Interpretability. Strict ordering allows for better interpretability. For example, we can infer why a certain machine was not chosen for a particular VM request. More broadly, we can aggregate statistics to determine how aggressive each rule is (e.g., what fraction of objects it "filters on average"). This evaluation helps us adjust the quantization of rules and their ordering if needed; see §6 for an example.

4 Architecture

In this section, we provide a high-level overview of Protean, and describe how the AA handles a service request.

Protean operation. Fig. 7 describes the high-level system architecture of Protean. Protean employs multiple Allocation Agents (AAs) that operate concurrently, following an optimistic concurrency model. The AAs are organized to run in multiple machines. Each machine hosts a single process, which in turn creates multiple worker threads, one thread per AA. Allocation requests from clients are routed to these processes through a load-balancer. Within a process, the requests are stored in a shared work-queue until they are picked up and processed by a free AA. The number of AAs is determined according to the peak instantaneous demand in the zone, while the number of AAs per machine depends on the memory footprint of each AA. Each AA makes allocation decisions based on its own (possibly stale) view of the inventory, and after processing a request successfully, tries to commit the result to a replicated store. The replicated store performs conflict detection, and serializes the commits to the same node (commits to different nodes are handled in parallel). Further, it stores all inventory information that is modified by the VM placement decisions from AAs. The replicated store functions as the authoritative source for the latest placement-related inventory state, and publishes all changes through a publish-subscribe (pub/sub) service.

Changes in the inventory that are not influenced by placement decisions, such as changes in machine health or capabilities, are also published via the pub/sub service. The AAs learn about inventory changes primarily through the updates produced by the pub/sub service. Additionally, on commit failures due to conflicts, they learn about the latest placement-related information for the conflicting machines as part of the failure notification.

Algorithm 1: Service allocation algorithm

```
1  def ALLOCATE_SERVICE(v_1,...,v_n, retries):
2      v_1,...,v_n ← ORDER(v_1,...,v_n)
3      for i = 1 to n do
4          m_i = ASSIGN_MACHINE_TO_VM(v_i)
5          if IS_INVALID_MACHINE(m_i) then
6              return FAILED
7      if COMMIT(m_1,...,m_n) then
8          return SUCCEEDED
9      else if retries < MAX_RETRIES then
10         return QUEUE FOR RETRY
11     else
12         return FAILED
```

Service allocation workflow. A service request may consist of multiple VM requests that are processed sequentially by a single AA. Algorithm 1 summarizes how Protean handles a service request. ORDER determines the order in which the VM requests will be processed. The goal of the ordering is to minimize the risk that a request is rejected due to fault domain considerations. ASSIGN_MACHINE_TO_VM attempts to assign a machine to a single VM by applying the rule logic; it is applied sequentially for each of the requested VMs (see §5 for implementation details). If the AA succeeds in assigning machines for all of the requested VMs, COMMIT tries to commit the service allocation result to the authoritative store. The commit fails if any of the VM-Machine assignments is invalidated because of a conflicting assignment made by another AA. On commit failures, the allocator state is rolled back and the entire request is re-queued for retry. The number of retries is configurable. We allow for a relatively high number of retries (more than 10) to avoid unnecessary allocation failures; however this has a negligible effect in production (e.g., the 99.9-percentile allocations succeed after three retries). The commit stage is pipelined with the previous stages, so that the AA is free to process the next request while a commit is in flight.

5 Protean Implementation

In this section, we describe our caching framework, which substantially expedites the ASSIGN_MACHINE_TO_VM pro-

cedure (§5.1–5.2). We also discuss our flexible conflict detection and reduction mechanisms (§5.3).

5.1 Preliminaries

To make high quality assignments, the AA initially considers the entire set of machines in the inventory as candidates.

Cluster selection. As discussed in §3.2, the AA starts the selection process by filtering and sorting clusters instead of machines. Since a zone has at most a few hundred clusters today, filtering and sorting the clusters is very fast (a couple of milliseconds at most). Accordingly, the cluster selection phase does not require any additional enhancements (such as caching past selection decisions). The output of this phase is the best 8 to 16 clusters; their machines (typically 10-15k) are the candidates for the machine selection process.

Machine selection – basic complexity. Recall that the machine-selection logic first trims the set of candidate machines to the set of valid machines by evaluating all validator rules for each machine. Then, it builds a comparison-based total ordering of the machines in the valid set, based on the suitability of each machine in hosting the VM (§3.3). Finally, a machine is randomly selected from the set of best machines. Building an *evaluation result* – the ordered list of valid machines – incurs a runtime complexity of $N\sum_{i=1}^{K_1} T_v(i) + N\log N\left(\sum_{i=1}^{K_2} T_p(i)\right)$, where N is the number of candidate machines, K_1 the number of validator rules, K_2 the number of preference rules, $T_v(i)$ the time to compute IsValid for the i^{th} validator rule, and $T_p(i)$ the time to compute Compare for the i^{th} preference rule. If the AA attempted to build this evaluation result from scratch for every request, it would exceed the required latency bounds for anything more than just a couple of thousand machines.

Motivation for caching. First, we observe *"Locality in requests"*. Each VM request is characterized by a vector of *trait* values. Example traits include: VM-Type, priority, and RequireIsolation (i.e., the VM should be on a machine of its own). There are tens of traits, each of which can take several values (including a "don't care" or empty option). In Sec. §2.2 we show that requests exhibit "locality" when zooming in on a single dimension (VM type). We note that this phenomenon carries over to the entire vector: there are a few value vectors commonly used across multiple requests, especially if they are chronologically close. The second observation is that the *inventory state changes slowly*. The state of a machine can change because of allocation-related events (addition, suspension, or deletion of VMs), or because of changes in health or other conditions of a VM or a machine. However, allocation-related events are the dominant reason for such changes. Hence the machines that change between consecutive executions of the AA are primarily the machines whose states were altered as a result of allocation decisions made by other AAs running in parallel. Since the number of parallel AAs is relatively small, there are typically not many such changes. These characteristics would allow us to drastically reduce the amount of computation performed in an execution of the machine selection logic by caching and reusing an evaluation "state" from previous executions. Intuitively, the only computation that is required is to update the state to incorporate the impact of inventory changes since the previous run. We next discuss the details of our caching approach.

5.2 Caching for Efficient Machine Selection

Each AA maintains a collection of cached objects, which together hold the information required for machine selection.

5.2.1 Caching Rule State

Caching internal rule state for efficient execution. Rules are the basic building blocks of the selection process. So, first and foremost, we use caching to improve the execution time for IsValid and Compare methods of the rules ($T_v(i)$ and $T_p(i)$ respectively). Specifically, every *rule type* implements these methods. The instantiations of each rule type, termed *rule objects*, are cached for reuse. A newly created rule object computes and stores all the information that it requires to execute the IsValid or Compare method in constant time. Usually this information is stored on a per machine basis. For example, the PreferNonEmptyMachines rule (see §3.2) stores a <*MachineID, Boolean*> dictionary that tracks whether each machine is empty or not.

Just-in-time updates of rule state. Every time a cached rule object is used, its internal state has to be brought up-to-date before its IsValid or Compare method can be called. To that end, along with the IsValid or Compare method, each rule implements the $Update(x_1,\ldots,x_m)$ method in order to update its stored state. The *Update* method is called immediately before the rule object is used. Its argument, $(x_1,\ldots,x_m)$, represents the latest state for machines that have *changed* from the last time the object was updated. Every rule can execute its IsValid or Compare function in constant time once it has updated its state with the latest changes.

Splitting rule state into multiple objects. The stored state of a rule object may depend on one or more request traits. For example, the AreNodeResourcesValid rule depends on the requested VM-Type, and hence must cache the Boolean whether the machine has enough capacity for each <*Machine, VM-Type*> pair. We observe, however, that to process a particular VM request, the rule object only needs the information for the VM-Type value of that request. Updating the state for every other VM-Type value would increase the just-in-time update time, and in turn the request processing time. Hence, instead of creating a single rule object for all requests, a rule object is created on demand for every VM-Type value. A rule object for a particular VM-Type value is used for all requests asking for that value. In effect, requests are divided into *equivalence classes* based on the relevant trait value, and

Time	Request	Cached Objects	Hits	Misses (Create New)
T1	$req(x_1,y_1)$	None	None	$Eval(x_1,y_1)$ $R_1(x_1,y_1)$ $R_2(x_1)$
T2	$req(x_1,y_2)$	$Eval(x_1,y_1)$ $R_1(x_1,y_1)$ $R_2(x_1)$	$R_2(x_1)$	$Eval(x_1,y_2)$ $R_1(x_1,y_2)$
T3	$req(x_1,y_1)$	$Eval(x_1,y_1)$ $Eval(x_1,y_2)$ $R_1(x_1,y_1)$ $R_2(x_1)$ $R_1(x_1,y_2)$	$Eval(x_1,y_1)^*$	None

Table 4: Example cache timeline. *If $Eval(x_1,y_1)$ needs to be updated then $R_1(x_1,y_1)$ and $R_2(x_1)$ would also be requested (and have cache hits).

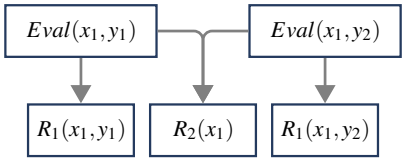


Figure 8: Cache hierarchy for the example in Table 4.

a single rule object handles all requests belonging to a class. For rules depending on multiple request traits, each unique combination of the trait values define a new equivalence class of requests for that rule. For the special case where a rule does not depend on any request trait (e.g., PreferNonEmptyMachine), a single rule object is used for all requests. This rule object specialization technique substantially reduces processing latency, and further decreases the effective memory size needed for caching.

Caching rule objects. Rule object references are stored in a constant size pool. The size is determined through trial-and-error based on memory footprint and hit-rate considerations. A rule object is identified by its type and the request trait value combination that it is associated with. Rule objects are evicted from the pool either if it is full (following a standard LRU eviction policy), or if they reach a certain age. Age-based eviction allows us to reduce the memory footprint during periods of low load.

5.2.2 Caching Rule Evaluation State

Caching the rule objects helps in substantially reducing $T_v(i)$ and $T_p(i)$. However, without any additional enhancements, we would still pay the sorting complexity of $N \log N$. Hence, we introduce additional objects termed *RuleEvaluation* objects. A rule evaluation object essentially holds the complete state of the evaluation, for a specific vector of trait values (see §5.1). The state includes the evaluation result (sorted list of machines) and references to relevant rule objects whose trait values match the respective values in the entire vector of trait values. A RuleEvaluation object is created after computing the evaluation result for a new vector of trait values, which serves as the identifier for the object). The object will then be reused for all requests that map to this identifier.

Updating the RuleEvaluation object. Similar to the rule objects, a cached RuleEvaluation object is brought up-to-date before it is used. However, unlike rule objects, RuleEvaluation objects use a common *Update* method, whose goal is to update the evaluation result with the changed machines. The method proceeds as follows: (1) the cached rule objects are brought up-to-date by calling their *Update* methods; (2) the modified machines are removed from the evaluation result; (3) the validator rules are run for each of these machines to determine which machines are valid; (4) valid machines are inserted back into the ordered list with an updated position. Because each insertion takes $\log N$ time, the *Update* method has runtime complexity of $M \log N$, where M is the number of machines with modified state. This is a substantial reduction in complexity, because $M \ll N$ (M is in the order of tens). RuleEvaluation objects are cached in another constant size memory pool, with an LRU eviction policy.

Example. Consider an example scenario where each request can have two traits $X \in \{x_1,x_2\}$ and $Y \in \{y_1,y_2\}$; and the allocation logic is expressed through two rules: R_1, which depends on traits X and Y, and R_2, which depends on X. Figure 8 shows the various rule and RuleEvaluation objects that are created and reused as the allocation engine serves incoming requests with different trait values. The accompanying Table 4 shows the hierarchy of objects that are created and cached as a consequence of processing the requests. The first request at time T1 has trait values $X = x_1$ and $Y = y_1$, and accordingly two new rule objects $R_1(x_1,y_1)$ and $R_2(x_1)$, and an evaluation object $Eval(x_1,y_1)$ are created. The second request at time T2 uses a different value y_2 for trait Y, and hence cannot reuse $Eval(x_1,y_1)$ or $R_1(x_1,y_1)$ objects. It reuses $R_2(x_1)$ since the trait value for X does not change, and creates new objects $R_1(x_1,y_2)$ and $Eval(x_1,y_2)$. The third request at time T3 uses the same trait values as the first, and hence is able to reuse all three objects that were created during the processing of the first request. Overall, two RuleEvaluation objects are created, corresponding to the two trait value vectors $\{x_1,y_1\}$ and $\{x_1,y_2\}$. They share a single object for rule R_2, but use two separate objects for rule R_1.

5.2.3 Additional Cache Hierarchies

Multiple rules often depend on the same part of the state. For example, multiple rules need to track whether machines are empty (e.g., BestFit and a rule that attempts to balance capacity across racks). For such cases, we encapsulate the shared part of the state in a *Shared-Cache* type, which multiple rules can refer to. Just like a Rule, a Shared-Cache implements the *Update* method, and declares any request traits it depends on. Shared-Caches play a huge role in reducing memory usage. These objects are cached in their own constant size memory pool. As mentioned, a cached object may depend on other

cached objects. The rule selection engine thus ensures that all dependencies are updated before the object is updated. Our cache hierarchy embeds desirable properties. For example, when a new RuleEvaluation object is created, it may often rely on existing rule and shared-cache objects.

5.2.4 Efficiently Updating the Cache

Tracking and updating mechanisms. Recall that each AA maintains its own private caches. Since each cached object can in principle be updated just before it is used, objects can exist at different levels of staleness. To facilitate seamless updates, each AA has a *journal* that keeps track of changes to any machine in the inventory. The journal maintains a global revision number which is incremented upon every update. In addition, the journal stores only the latest machine state for every machine. Every cached object stores the highest revision number it has seen, which corresponds the latest inventory update it has consumed. An object brings itself up to date by reading only the journal records with higher revision numbers. Consequently, the update operation has runtime complexity at the order of the number of machines that were *modified*, rather than the entire inventory. The AA updates the journal during an ongoing evaluation to record each VM placement decision that it is making. In addition, the AA updates the journal between evaluations by processing enqueued incoming changes from the pub/sub service or the placement store.

Background updates. An up-to-date cache can handle a request in few ms by simply extracting the best machine(s). However, when this is not the case, just-in-time cache update times can be a significant part of the total VM request latency. Nonetheless, because the system has multiple AAs that are provisioned to handle the rare periods of peak load, they remain inactive for most of the time. Hence, when an AA has no requests to process, it is used to opportunistically update the caches (starting with RuleEvaluation cache objects and proceeding recursively).

5.2.5 Discussion

Design advantages. One clear advantage of our caching approach over other alternatives (e.g., node sampling or strict partitioning the inventory) is that Protean can sustain low latency and high throughput without giving up on allocation opportunities. Another appealing property of our implementation is that the complexity of creating, reusing, and updating a rule object is almost completely hidden from the creator of a rule. A rule only has to implement the IsValid (or Compare) and Update methods and declare the request traits it depends on. The rest is handled by the machine selection engine within the AA. This clear separation between the rules and the evaluation engine has been instrumental in the extensibility and adaptability of Protean.

Global rules. There are a few machine selection rules that do not express preference for individual machines, but rather among groups of machines (e.g., prefer the least used rack). We refer to such rules as *global* rules. Most global rules reason about clusters and hence are part of cluster selection. However, a few rules also reason about racks, and hence are part of the machine selection stage. Global rules require us to adjust our caching methodology. To understand the issue, observe that for such rules, a change in a single machine within the group might impact the value of all other machines in that group (e.g., an allocation to a single machine in a rack might make the rack less attractive than another rack). Hence, a single machine change makes *all* machines in the group 'dirty', and the cached objects would require updating all the machines in the group. To still benefit from our caching infrastructure, we use a divide and conquer approach: in a nutshell, we divide the machine inventory into *cells*. Each cell consists of a subset of machines who are considered identical from the perspective of all the global rules. We apply our caching mechanisms separately for each cell; that means that we maintain a sorted list of valid machines (the filtering and sorting is done based on all non-global rules). To obtain the actual evaluation result, we pick the best machine from each cell, and do the required comparisons and sorting based on all rules. While these comparisons slow the evaluation time, we note that the original complexity term of $N \log N$ reduces to $N_c \log N_c$, where N_c is the number of cells. In our current setting, cells correspond to racks. The number of racks after cluster selection is in the order of a hundred, hence $N_c \ll N$.

5.3 Conflict Detection and Reduction

Occasional spikes of thousands of requests push all AAs to work at full tilt. Naturally, chances of commit failures due to conflicts increase considerably during such periods. Conflicts reduce the effective throughput and increase outright failures; as a request fails after a fixed number of retries. We employ the following strategies to reduce such failures.

Fine-grained conflict detection. We built a conflict-detection mechanism which allows commits to succeed even when the AA makes a placement decision based on an out-dated view of a machine. The logic verifies that the new placement decision does not over-commit the machine resources or violate other anti-colocation constraints (such as placing a new VM on a machine that already hosts a VM requiring isolation). If so, it merges the new placement decision with the current state of the machine as part of the commit. This mechanism has led to 25% drop in conflicts in one of our busiest zones, compared to the simpler strategy of rejecting all out-dated placement decisions.

Trading allocation quality for conflict reduction. Conflicts increase during high-load periods, not only because of rapid inventory changes, but also because AAs apply the same logic; AAs are likely to identify highly overlapping sets of best ma-

chines if their respective requests are similar. The number of machines in this set might be very small, so that even a random selection from it may lead to a conflict with high probability. To address this challenge, the AA employs an hybrid strategy for selecting a machine in its final step. In periods with no conflicts, it selects from the set of best machines. Alternatively, it may use a more permissive *conflict-avoidance* scheme; this scheme randomly selects a machine from the top N_0 machines, where N_0 is a configurable parameter (note that the top N_0 machines may differ in their desirability). The conflict-avoidance scheme is enabled with a probability proportional to the ratio of conflict failures to total commit attempts, measured using a rolling window of the most recent commit attempts. The conflict-avoidance scheme is instrumental in satisfying demand at high-load periods; since such periods are infrequent, it has little effect on allocation quality.

6 Evaluation

6.1 Methodology

Production measurements. Since Protean is fully deployed across all regions of Azure, it is natural to evaluate it using measurements from production. Our infrastructure collects numerous diagnostic metrics and structured logs, which are used for monitoring and evaluation. These metrics and logs are aggregated into a central and easily accessible source, which allows custom queries for specific data extraction. Production measurements is the default method in our evaluation; we will mention explicitly when we use simulations.

Simulations. Simulations are incredibly useful for evaluating what-if scenarios, such as the effect of different inventory sizes, different rule configurations, etc. Our simulations use real traces and configurations as input, and can be considered a reasonably accurate representation of reality. In particular, the simulated workload includes both traces from production, as well as realistic probabilistic models of VM requests, derived from historical traces. We built two types of simulators. The *high-fidelity* simulator uses the actual production code of Protean to perform the allocations, and outputs large amounts of data for debugging purposes. Our *low-fidelity* simulator includes a lightweight emulation of the allocator (e.g., supports a subset of the rules). This simulator still provides an excellent approximation of the system, is orders-of-magnitude faster, and especially useful for large scale evaluations.

6.2 Performance and Scale

Here we evaluate key mechanisms and design choices that help Protean scale. We focus on the caching mechanism and the effect of multiple AAs.

Cache evaluation: Hit-rate, latency and update overhead. As discussed in §2, the nature of our workload motivates the use of caching. Our first goal is to understand the effectiveness of the hierarchical cache architecture. Towards that end, Fig. 9 shows typical hit-ratio patterns in one of our zones, focusing on four different cacheable classes over a day period (taking into account all cached objects for each class). The FdEmptyNodes and BestFit are cached rules; *NodeResources* is a Shared-Cache; and *RuleEvaluation* corresponds to the RuleEvaluation class. The stacked bar to the right of the figure shows the frequency of cache requests as a percentage of allocation requests (NodeResources cache is requested only in 0.16% of allocations, hence barely noticed). Lower-level caches are only requested when higher-level caches miss, so to interpret the results, both the request frequency and hit-ratio should be considered. For example, the NodeResources cache has a hit-ratio around 20%, but is less frequently accessed – compared to RuleEvalution, which has a much higher hit-ratio and is accessed on every request.

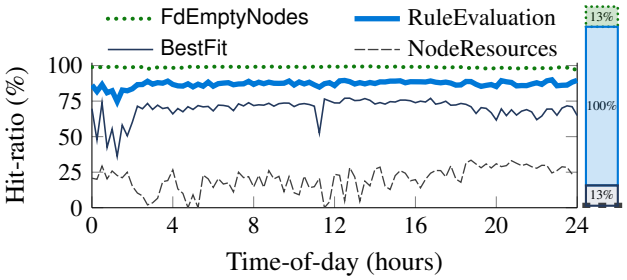


Figure 9: Cache-hit ratio over time for some caches. The frequency of requests for each cache is shown on the right.

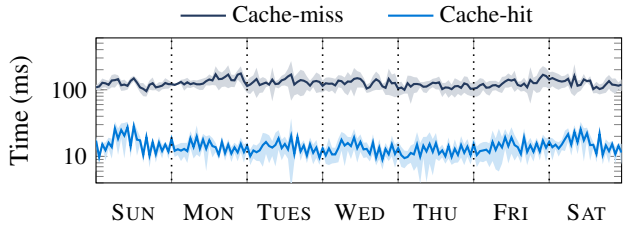


Figure 10: Latency in a large zone; average (+standard deviation) per hour per day, over two months.

The resulting benefit of our cache and high hit-rates for evaluation caches is improved latency. This can be clearly seen from Fig. 10, which depicts the effect of a cache hit/miss for the RuleEvaluation cache. Given our high hit-rate, the overall average latency is close to 20 ms per allocation. A cache-miss still uses many lower-level caches so that the latency is typically 70-80ms. We note that Protean's latency is affected by additional functionality beyond the allocation process itself, such as tracking and outputting debug information about every allocation.

To gain further insights into the cache operation and resulting latencies, we track the average number of machines updated per allocation in one of our zones (∼30k machines), over an entire day. Cache hits/misses have a significant impact on the number of updated machines: approximately 50 for a

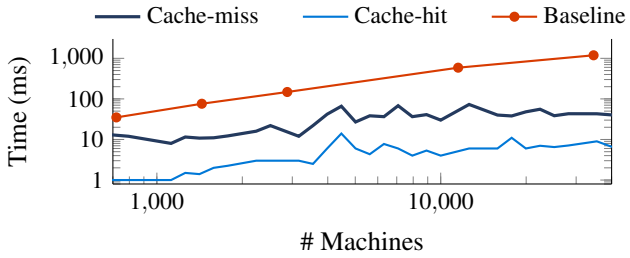


Figure 11: Median latency vs inventory size over one-month. Baseline results are based on high-fidelity simulation.

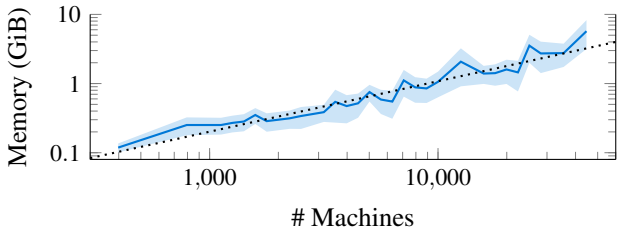


Figure 12: Memory used per AA vs inventory size. Average (+standard deviation) over a month over all zones.

hit verses 2000 for a miss. A hit thus translates to less overhead in the allocation process and in turn, to lower latency. Notably, the number of updates is relatively small even in case of a miss (6.6% of the machines). This can be attributed to two main factors. First, the cluster selection process filters out a large part of the inventory. In particular, we observe through simulations that cluster selection rules filter out up to 20% of the inventory for zones of 10k machines or more. Second, a substantial part of inventory updates occur asynchronously via background resolve. Indeed, our production measurements indicate that 80-96% of machine updates are done by that mechanism.

Scaling with inventory. Fig. 11 shows the inventory size effect on latency under three different scenarios. The first two scenarios correspond to cache hit or miss for RuleEvaluation cache, where the results represent production measurements of zones with different sizes. To further examine the effectiveness of mechanisms, we include a third scenario ("baseline") where the caching and cluster selection are disabled; because we would rather not disable these mechanisms in production, we use our high-fidelity simulator to obtain the results; observe that the median latency reaches around 1000 ms for larger zones. In addition to latency performance, it is also important to examine the cache's memory footprint. Fig. 12 shows the memory required per AA, as a function of inventory size. The fitted line, obtained via regression, shows that the growth of our the memory footprint is sublinear ($\sim x^{0.73}$), which helps keep memory sizes manageable at scale.

Multiple allocation agents. Multiple allocators influence important metrics, such as the number of conflicts, delay and

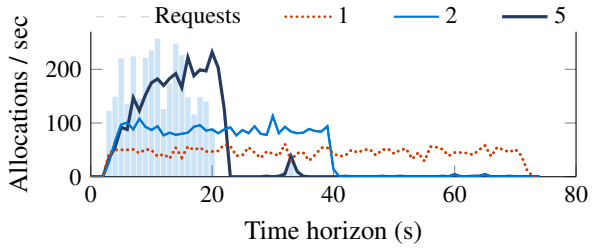


Figure 13: [Simulation] Throughput (allocations per second) over a selected time horizon, for various number of AAs. The bars represent the requests, and the curves represent their processing. A shorter x-axis "span" means better performance.

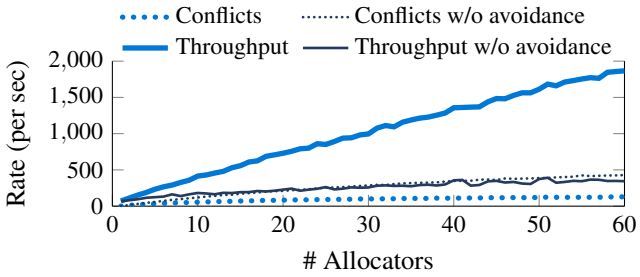


Figure 14: [Simulation] 99th percentile of conflicts per second and corresponding throughput. Uses production trace as input.

throughput. Since the number of AAs in production is fixed, we use our low-fidelity simulator for the experiments: we emulate an heterogeneous inventory of one of our zones with nearly 25k machines. We replay an actual request trace of an entire day. We use the same conflict-avoidance (see §5.3) parameters (100 machines allowed for final random selection, rolling window size of 50 commits) and number of retries before rejection (20) as in production. To expedite the low-fidelity simulations, the actual cache infrastructure is not integrated in the simulator. To mimic the cache, we use a realistic statistical model, derived from production measurements of the same zone over a month period. A cache hit/miss is determined using a Bernoulli random variable, with an average hit-ratio ($p = 0.9$) obtained from production; a hit results in 14ms latency, whereas a miss incurs a higher latency of 88ms.

Our first experiment examines how different number of AAs handle a spike in demand (see Fig. 13). The key takeaway here is that a single allocator struggles to satisfy the load in a reasonable time, causing excess delays to requests. With five AAs, the requests are handled in more than 3x less time (adding more AAs yields similar results). Our second experiment (Fig. 14) replays another trace from the same zone; the figure depicts the 99th percentile for the conflicts per second, as well as the throughput observed during the same time. Our collision-avoidance strategy provides clear gains: note that throughput increases substantially with the number of AAs with little effect on conflicts.

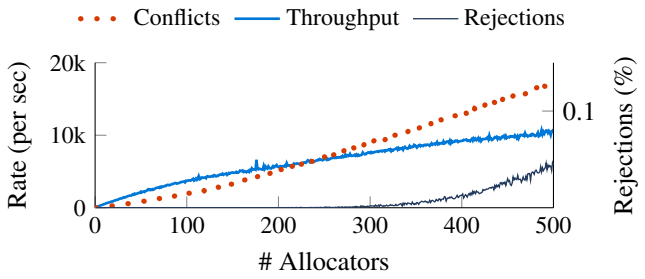


Figure 15: [Simulation] Scale test. 99th percentile of conflicts per second with corresponding throughput and rejection rate.

Our system easily handles the request volume currently seen in production. It is of natural interest to study (via simulations) the possible performance trends at higher scale. Towards that end, we take an existing request trace and speed up time by factors of up to 1000x. Fig. 15 shows throughput, conflicts and rejection rates as we scale up the demand, verses the number of AAs handling the requests; the simulations use our standard avoidance scheme. We observe that although throughput reaches over 10k requests per second, this comes at the expense of significant conflicts which in turn affect the rejection rate. The throughput eventually plateaus, likely due to a combination of fixed inventory size and increasing conflicts. In a production setting, we would have several options to deal with increased demand (e.g., tune the avoidance scheme, or longer-term increase of inventory size).

6.3 Allocation Quality

Quality vs. performance tradeoff. There are different criteria for quantifying quality; for example, balancing allocations across multiple fault domains is important for satisfying large service requests. In this section, we zoom in on a key efficiency metric – *packing density* – which measures the average number of allocated cores on non-empty machines (certain machines must be kept empty, e.g., for failover of large VMs). Formally, the packing density at time t is the ratio between the number of allocated cores, and the number of non-empty machines times the number of cores in each machine. We note that packing density can be defined similarly for other resources, such as memory; we focus on CPU because it is typically the bottleneck resource. Table 5 summarizes a set of experiments in one of our zones, using the low-fidelity simulator on a 5 month trace. The different rows correspond to different parameter configurations of the BestFit rule; in particular, the configurations differ by the number of buckets (see §3.2), where ∞ means no quantization. Recall that the more buckets we use the finer is the quantization of the score, which allows for better discrimination of machines by the packing quality. On the flip side, a finer quantization means that downstream rules are left with fewer candidate machines. The results demonstrate some interesting trends. As expected, the packing density (denoted PD %) increases with the number of buckets; note that the most significant increase is from two to three buckets. More buckets increases the packing density by a little, however at the cost of filtering out a substantial percent of candidate machines (Post-BestFit). The effect is magnified at the extreme of no quantization, where very few candidate machines are left. As a consequence, not only downstream rules become meaningless, but also the conflict rate increases. This is because different allocators are more likely to pick the same machine for allocation. In view of the above analysis, we currently use three buckets in production.

Buckets	PD (%)	Post-BestFit (%)	P_{99} Conflicts / min
1	83.5	27.6	13.4
2	84.3	25.0	13.5
3	86.3	21.0	11.6
4	87.3	16.5	12.0
5	87.8	13.7	10.7
∞	89.1	2.3	18.0

Table 5: [Simulation] The trade-off between packing and robust allocations. PD (%) is the packing density averaged over five months.

Adapting to COVID-19 capacity crunch. As a consequence of the COVID-19 pandemic, Azure observed a sharp increase in demand. As an immediate response, we increased the utilization limits in each cluster by 1%. These limits are used to leave enough buffers for in-cluster scale-outs as well as to account for failures. The increase was done easily by modifying configurable threshold values in a cluster validator rule IsClusterBelowLimit(x,v). This limit change slightly increased the risk for scale-outs and fail-overs. To mitigate the risk, we used Protean to identify fragmented machines, and recommend migration targets that would improve packing (what-if analysis). A supplementary VM migration mechanism used these recommendations to live-migrate some VMs (targeting first-party VMs only), resulting in improved packing density. Fig. 16 shows both the average utilization (i.e., ratio between number of allocated cores and total number of cores in Azure) and the packing density over our entire fleet. The dashed lines indicates the point of time at which the above changes were made. The net effect of Protean adaptation was a sizeable increase in utilization, facilitated by a significant improvement in packing density. We also depict in the same figure the relative trends for the overall capacity fulfillment rate (CFR) – the fraction of allocations that are successfully deployed. CFR dipped slightly in mid-March, but went up again exceeding its target of four nines by mid-April.

7 Related Work

Resource management for large compute clusters. Numerous systems have been implemented for various domains, including batch scheduling for HPC applications

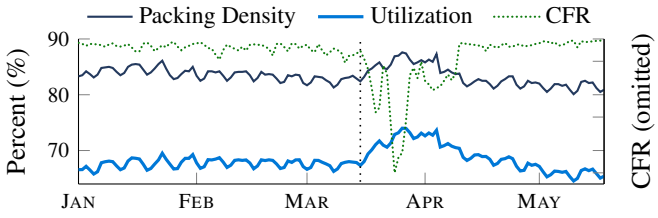


Figure 16: CPU usage across Azure. Absolute capcity fulfillment rate (CFR) values are omitted for confidentiality (hence, the CFR y-axis is not labeled). The curve represents the relative trend and scaled to fit the graph.

[25, 44, 45], big-data analytics [7, 20, 23, 28, 38, 40, 47, 52], stream-processing [34], AI [36], etc. More related to our context is the work on hyper-scale cloud computing clusters, see [8, 12, 15–17, 43, 48] and references therein. Our cloud workload analysis adds to a body of work on this topic, e.g., [5, 10, 12, 18, 26, 30, 35, 42, 46, 50].

Scheduler types. One useful way to classify large-scale schedulers is based on how they process work items (jobs, tasks, VMs, etc.). *Centralized monolithic* schedulers [21, 24, 52] use a single agent to process requests. They avoid concurrency issues, yet are harder to scale. A subset of these schedulers, optimizes placement decisions by *batch-processing* multiple jobs together [19, 21, 24]. Our demanding latency and throughout requirements preclude using these approaches. To cope with scale and management complexities, *two-level* schedulers [23, 47] perform course-grained resource management, while leaving the fine-grained scheduling to application frameworks. Similarly, *distributed schedulers* [36, 38, 41] decentralize the scheduling logic by employing sophisticated queue management strategies at the target machines (see also works on *hybrid schedulers* [13, 14, 29]). Two-level or various distributed approaches are less applicable for VM scheduling, which is inherently IaaS-centric. Our AA is centralized, while target machines create their assigned VMs according to a simple FCFS policy.

Concurrent schedulers. Similar to Sparrow [38], Apollo [7] and Omega [43], Protean is a concurrent scheduler which employs multiple agents over a shared inventory. Omega handles conflicts immediately as part of scheduling, whereas Apollo and Mercury allow conflicting scheduling decisions to queue on target nodes while deferring conflict resolution. As in [43], we use multiple concurrent allocation agents and a conflict resolution model. Indeed, our customers prefer VM requests to fail early rather than waiting longer in hope for success; this allows higher level services to quickly try other alternatives, such as using another zone or modifying some request properties.

Allocation scope. Cluster selection and caching allow Protean to make resource assignment decisions based on the entire inventory, similar to [7, 15]. Alternatively, schedulers can statically partition the inventory [49], or use random sampling to make a decision using a subset of the inventory [17, 38, 48]. Protean shares similarities with Google's Borg [48]. Borg employs other optimizations for scalability, such as caching node preference scores until the node changes, and avoiding duplicate work by evaluating decisions for only a single task within a group of identical tasks. Protean caches not only node-centric data, but also rule and evaluation outcomes that can be used across different requests. In addition, Borg introduces the notion of equivalent classes, where feasibility and scoring is determined only for a single task out of identical tasks in a job. Protean extends this idea by considering requests across tenants to be equivalent if they share the same trait values. Finally, unlike Borg, we do not employ sampling (termed "relaxed randomization"), but rather use other techniques to help with scale (multi-layer caching and cluster selection).

Resource efficiency. Cloud schedulers attempt to increase actual resource usage through a variety of techniques, e.g., reclaiming unused resources, harvesting, profiling, heterogeneity and interference awareness [9, 12, 15, 16, 22, 27, 32, 33, 37, 48, 51, 53]. Protean's flexible rule-based logic facilitates dynamic resource adjustment and interference mitigation strategies; their description is outside the scope of this paper.

8 Conclusion

We describe Protean, the VM allocation service of Azure. Our design separates policy from mechanisms, which has allowed us to successfully expand our VM offerings over the years. A flexible rule-based allocator facilitates refining the allocation logic and explaining it to customers. VM requests are processed in milliseconds, due to a hierarchical caching framework. Results from production demonstrate that Protean sustains adequate trade-offs between performance and quality.

Acknowledgements

We are grateful to our shepherd, John Wilkes, and the anonymous reviewers for their detailed and thoughtful feedback. We would like to acknowledge the contributions of the engineers who have been involved in the design, implementation, and maintenance of Protean over the years - Jason Chu, Chris Cowdery, Dustin Dobransky, Eric Hao, Ryan Hidalgo, Valentina Li, John Miller, Mukund Nigam, Jason Seo, Kanishk Thareja, Karel Trueba, Yiran Wei and Brian Yan. We also thank Saurabh Agarwal, Girish Bablani, Ricardo Bianchini, Íñigo Goiri, Marcus Fontoura and Saad Syed for helpful discussions.

References

[1] Azure batch, 2020. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms.

[2] Azure spot virtual machines, 2020. https://azure.microsoft.com/en-us/pricing/spot/.

[3] Azure VM packing trace (public dataset), 2020. https://github.com/Azure/AzurePublicDataset.

[4] Azure's virtual machine scale sets, 2020. https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-autoscale-overview.

[5] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *USENIX Annual Technical Conference (ATC)*, pages 533–546. USENIX Association, July 2018.

[6] Yossi Azar and Danny Vainstein. Tight bounds for clairvoyant dynamic bin packing. *ACM Trans. Parallel Comput.*, 6(3):15:1–15:21, 2019.

[7] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–300, 2014.

[8] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Queue*, 14(1):70–93, January 2016.

[9] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term SLOs for reclaimed cloud computing resources. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 1–13, 2014.

[10] Yue Cheng, Zheng Chai, and Ali Anwar. Characterizing co-located datacenter workloads: An Alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*, 2018.

[11] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. Dynamic bin packing. *SIAM Journal on Computing*, 12(2):227–258, 1983.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 153–167, 2017.

[13] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*, page 497–509, 2016.

[14] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC)*, page 499–510, 2015.

[15] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 77–88, 2013.

[16] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 127–144, 2014.

[17] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 97–110, 2015.

[18] Sheng Di, Derrick Kondo, and Walfredo Cirne. Characterization and comparison of cloud versus grid workloads. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing (CLUSTER)*, page 230–238, 2012.

[19] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[20] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *2015 USENIX Annual Technical Conference (ATC)*, pages 459–471. USENIX Association, July 2015.

[21] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 99–115, 2016.

[22] Jaeung Han, Seungheun Jeon, Young-ri Choi, and Jaehyuk Huh. Interference management for distributed parallel applications in consolidated clusters. In *Proceedings of the Twenty-First International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS), page 443–456, 2016.

[23] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 22–22, 2011.

[24] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, page 261–276, 2009.

[25] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the Maui scheduler. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, page 87–102. Springer-Verlag, 2001.

[26] C. Jiang, G. Han, J. Lin, G. Jia, W. Shi, and J. Wan. Characteristics of co-allocated online services and batch jobs in internet data centers: A case study from Alibaba cloud. *IEEE Access*, 7:22495–22508, 2019.

[27] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[28] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated SLOs for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI*, pages 117–134, 2016.

[29] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC)*, page 485–497, 2015.

[30] Cinar Kilcioglu, Justin M. Rao, Aadharsh Kannan, and R. Preston McAfee. Usage patterns and the economics of the public cloud. In *Proceedings of the 26th International Conference on World Wide Web (WWW)*, page 83–91, 2017.

[31] Alok Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. Prediction-based power oversubscription in cloud platforms, 2020.

[32] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, page 619–630, 2013.

[33] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, page 248–259, 2011.

[34] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. Turbine: Facebook's service management platform for stream processing. In *Proceedings of the 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020.

[35] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads: Insights from Google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37(4):34–41, March 2010.

[36] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 561–577, 2018.

[37] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, page 184–200, 2017.

[38] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, page 69–84, 2013.

[39] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. Technical report, 2011. Available from https://www.microsoft.com/en-us/research/wp-content/uploads/2011/01/VBPackingESA11.pdf.

[40] Hang Qu, Omid Mashayekhi, David Terei, and Philip Levis. Canary: A scheduling architecture for high performance cloud computing. *arXiv preprint arXiv:1602.01412*, 2016.

[41] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[42] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC)*, 2012.

[43] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 351–364, 2013.

[44] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, page 8–es, 2006.

[45] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[46] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, pages 1–14, 2020.

[47] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*, pages 1–16, 2013.

[48] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, pages 1–17, 2015.

[49] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. Pigeon: An effective distributed, hierarchical datacenter job scheduler. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, page 246–258, 2019.

[50] John Wilkes. More Google cluster data. Google research blog, Nov 2011. Available from https://ai.googleblog.com/2011/11/more-google-cluster-data.html.

[51] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, page 607–618, 2013.

[52] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, page 265–278, 2010.

[53] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 755–770, 2016.



Ansor: Generating High-Performance Tensor Programs for Deep Learning

Lianmin Zheng [1], Chengfan Jia [2], Minmin Sun [2], Zhao Wu [2], Cody Hao Yu [3],
Ameer Haj-Ali [1], Yida Wang [3], Jun Yang [2], Danyang Zhuo [1,4],
Koushik Sen [1], Joseph E. Gonzalez [1], Ion Stoica [1]

[1] *UC Berkeley,* [2]*Alibaba Group,* [3]*Amazon Web Services,* [4] *Duke University*

Abstract

High-performance tensor programs are crucial to guarantee efficient execution of deep neural networks. However, obtaining performant tensor programs for different operators on various hardware platforms is notoriously challenging. Currently, deep learning systems rely on vendor-provided kernel libraries or various search strategies to get performant tensor programs. These approaches either require significant engineering effort to develop platform-specific optimization code or fall short of finding high-performance programs due to restricted search space and ineffective exploration strategy.

We present Ansor, a tensor program generation framework for deep learning applications. Compared with existing search strategies, Ansor explores many more optimization combinations by sampling programs from a hierarchical representation of the search space. Ansor then fine-tunes the sampled programs with evolutionary search and a learned cost model to identify the best programs. Ansor can find high-performance programs that are outside the search space of existing state-of-the-art approaches. In addition, Ansor utilizes a task scheduler to simultaneously optimize multiple subgraphs in deep neural networks. We show that Ansor improves the execution performance of deep neural networks relative to the state-of-the-art on the Intel CPU, ARM CPU, and NVIDIA GPU by up to 3.8×, 2.6×, and 1.7×, respectively.

1 Introduction

Low-latency execution of deep neural networks (DNN) plays a critical role in autonomous driving [14], augmented reality [3], language translation [15], and other applications of AI. DNNs can be expressed as a directed acyclic computational graph (DAG), in which nodes represent the operators (*e.g.*, convolution, matrix multiplication) and directed edges represent the dependencies between operators. Existing deep learning frameworks (*e.g.*, Tensorflow [1], PyTorch [39], MXNet [10]) map the operators in DNNs to vendor-provided kernel libraries (*e.g.*, cuDNN [13], MKL-DNN [27]) to achieve high performance. However, these kernel libraries require significant engineering effort to manually tune for each hardware platform and operator. The significant manual effort required to produce efficient operator implementations for each target accelerator limits the development and innovation of new operators [7] and specialized accelerators [35].

Given the importance of DNNs' performance, researchers and industry practitioners have turned to search-based compilation [2, 11, 32, 49, 59] for automated generation of *tensor programs*, *i.e.*, low-level implementations of tensor operators. For an operator or a (sub-)graph of multiple operators, users define the computation in a high-level declarative language (§2), and the compiler then searches for programs tailored towards different hardware platforms.

To find performant tensor programs, it is necessary for a search-based approach to explore a large enough search space to cover all the useful tensor program optimizations. However, existing approaches fail to capture many effective optimization combinations, because they rely on either predefined manually-written templates (*e.g.*, TVM [12], FlexTensor [59]) or aggressive pruning by evaluating incomplete programs (*e.g.*, Halide auto-scheduler [2]), which prevents them from covering a comprehensive search space (§2). The rules they use to construct the search space are also limited.

In this paper, we explore a novel search strategy for generating high-performance tensor programs. It can automatically generate a large search space with comprehensive coverage of optimizations and gives every tensor program in the space a chance to be chosen. It thus enables to find high-performance programs that existing approaches miss.

Realizing this goal faces multiple challenges. First, it requires automatically constructing a large search space to cover as many tensor programs as possible for a given computation definition. Second, we need to search efficiently without comparing incomplete programs in the large search space that can be orders of magnitude larger than what existing templates can cover. Finally, when optimizing an entire DNN with many subgraphs, we should recognize and prioritize the subgraphs that are critical to the end-to-end performance.

To this end, we design and implement *Ansor*, a framework for automated tensor program generation. Ansor utilizes a hierarchical representation to cover a large search space. This representation decouples high-level structures and low-level details, enabling flexible enumeration of high-level structures and efficient sampling of low-level details. The space is constructed automatically for a given computation definition. Ansor then samples complete programs from the search space and fine-tunes these programs with evolutionary search and a learned cost model. To optimize the performance of DNNs with multiple subgraphs, Ansor dynamically prioritizes subgraphs of the DNNs that are more likely to improve the end-to-end performance.

We evaluate Ansor on both standard deep learning benchmarks and emerging new workloads against manual libraries and state-of-the-art search-based frameworks. Experiment results show that Ansor improves the execution performance of DNNs on the Intel CPU, ARM CPU, and NVIDIA GPU by up to 3.8×, 2.6×, and 1.7×, respectively. For most computation definitions, the best program found by Ansor is outside the search space of existing search-based approaches. The results also show that, compared with existing search-based approaches, Ansor searches more efficiently, generating higher-performance programs in a shorter time, despite its larger search space. Ansor can match the performance of a state-of-the-art framework with an order of magnitude less search time. Besides, Ansor enables automatic extension to new operators by only requiring their mathematical definitions without manual templates.

In summary, this paper makes the following contributions:

- A mechanism to generate a large hierarchical search space of tensor programs for a computational graph.
- An evolutionary strategy with a learned cost model to fine-tune the performance of tensor programs.
- A scheduling algorithm based on gradient descent to prioritize important subgraphs when optimizing the end-to-end performance of DNNs.
- An implementation and comprehensive evaluation of the Ansor system demonstrating that the above techniques outperform state-of-the-art systems on a variety of DNNs and hardware platforms.

2 Background

The deep learning ecosystem is embracing a rapidly growing diversity of hardware platforms including CPUs, GPUs, FPGAs, and ASICs. In order to deploy DNNs on these platforms, high-performance tensor programs are needed for the operators used in DNNs. The required operator set typically contains a mixture of standard operators (*e.g.*, matmul, conv2d) and novel operators invented by machine learning researchers (*e.g.*, capsule conv2d [23], dilated conv2d [57]).

Matrix Multiplication $C_{i,j} = \sum_k A_{i,k} B_{k,j}$

```
C = compute((N, M), lambda i, j: sum(A[i, k]*B[k, j], [k]))
```

Figure 1: The computation definition of matrix multiplication.

To deliver portable performance of these operators on a wide range of hardware platforms in a productive way, multiple compiler techniques have been introduced (*e.g.*, TVM [11], Halide [41], Tensor Comprehensions [49]). Users define the computation in a form similar to mathematical expressions using a high-level declarative language, and the compiler generates optimized tensor programs according to the definition. Figure 1 shows the computation definition of matrix multiplication in the TVM tensor expression language. Users mainly need to define the shapes of the tensors and how each element in the output tensor is computed.

However, automatically generating high-performance tensor programs from a high-level definition is extremely difficult. Depending on the architecture of the target platform, the compiler needs to search in an extremely large and complicated space containing combinatorial choices of optimizations (*e.g.*, tile structure, tile size, vectorization, parallelization). Finding high-performance programs requires the search strategy to cover a comprehensive space and explore it efficiently. We describe two recent and effective approaches in this section and other related work in §8.

Template-guided search. In template-guided search, the search space is defined by manual templates. As shown in Figure 2a, the compiler (*e.g.*, TVM) requires the user to manually write a template for a computation definition. The template defines the structure of the tensor programs with some tunable parameters (*e.g.*, tile size and unrolling factor). The compiler then searches for the best values of these parameters for a specific input shape configuration and a specific hardware target. This approach has achieved good performance on common deep learning operators. However, developing templates requires substantial effort. For example, the code repository of TVM already contains more than 15K lines of code for these templates. This number continues to grow as new operators and new hardware platforms emerge. Besides, constructing a quality template requires expertise in both tensor operators and hardware. It takes non-trivial research effort [32, 55, 59] to develop quality templates. Despite the complexity of template design, manual templates only cover limited program structures because manually enumerating all optimization choices for all operators is prohibitive. This approach typically requires defining one template for each operator. FlexTensor [59] proposes a general template to cover multiple operators, but its template is still designed for single operator granularity, which fails to include optimizations involving multiple operators (*e.g.*, operator fusion). The search space of optimizing a computational graph with multiple operators should contain different ways to compose the operators. A template-based approach fails to achieve this because it cannot break down their fixed templates and re-compose them

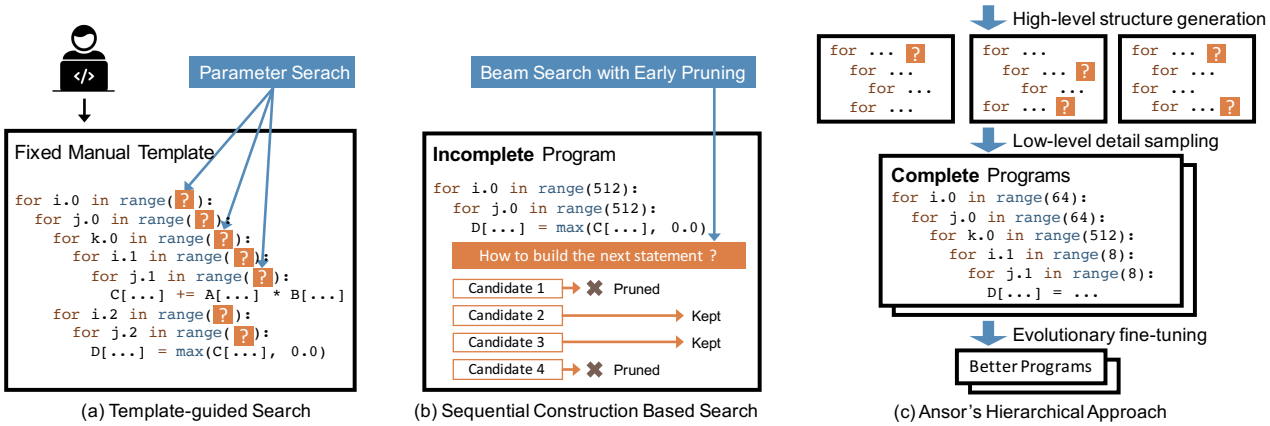


Figure 2: Search strategy comparison. The pseudo-code shows tensor programs with loop nests. The question marks in orange background denote low-level parameters.

during the search.

Sequential construction based search. This approach defines the search space by decomposing the program construction into a fixed sequence of decisions. The compiler then uses an algorithm such as beam search [34] to search for good decisions (*e.g.*, Halide auto-scheduler [2]). In this approach, the compiler constructs a tensor program by sequentially unfolding all nodes in the computational graph. For each node, the compiler makes a few decisions on how to transform it into low-level tensor programs (*i.e.,* deciding computation location, storage location, tile size, *etc.*). When all nodes are unfolded, a complete tensor program is constructed. This approach uses a set of general unfolding rules for every node, so it can search automatically without requiring manual templates. Because the number of possible choices of each decision is large, to make the sequential process feasible, this approach keeps only top-k candidate programs after every decision. The compiler estimates and compares the performance of candidate programs with a learned cost model to select the top-k candidates; while other candidates are pruned. During the search, the candidate programs are incomplete because only part of the computational graph is unfolded or only some of the decisions are made. Figure 2b shows this process.

However, estimating the final performance of incomplete programs is difficult in several respects: (1) the cost model trained on complete programs cannot accurately predict the final performance of incomplete programs. The cost model can only be trained on complete programs because we need to compile programs and measure their execution time to get the labels for training. Directly using this model to compare the final performance of incomplete programs will result in poor accuracy. As a case study, we train our cost model (§5.2) on 20,000 random complete programs from our search space and use the model to predict the final performance of incomplete programs. The incomplete programs are obtained by only applying a fraction of loop transformations of the complete programs. We use two ranking metrics for evaluation: the accuracy of pairwise comparison and the recall@k

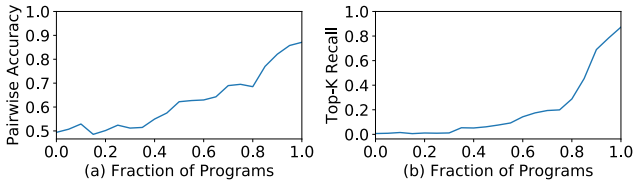


Figure 3: Pairwise comparison accuracy and top-k recall curve on random partial programs. In both subfigures, higher values are better.

score of top-k programs [1] ($k = 10$). As shown in Figure 3, the two curves start from 50% and 0% respectively, meaning that random guess with zero information gives 50% pairwise comparison accuracy and 0% top-k recall. The two curves increase quickly as the programs become complete, which means the cost model performs very well for complete programs but fails to accurately predict the final performance of incomplete programs. (2) The fixed order of sequential decisions limits the design of the search space. For example, some optimization needs to add new nodes to the computational graph (*e.g.*, adding cache nodes, using `rfactor` [46]). The number of decisions for different programs becomes different. It is hard to align the incomplete programs for a fair comparison. (3) Sequential construction based search is not scalable. Enlarging the search space needs to add more sequential construction steps, which, however, leads to a worse accumulated error.

Ansor's hierarchical approach As shown in Figure 2c, Ansor is backed by a hierarchical search space that decouples high-level structures and low-level details. Ansor constructs the search space for a computational graph automatically, eliminating the need to manually develop templates. Ansor then samples complete programs from the space and performs fine-tuning on complete programs, avoiding the inaccurate estimation of incomplete programs. Figure 2 shows the key dif-

[1] recall@k of top-k $= \frac{|G \cap P|}{k}$, where G is the set of top-k programs according to the ground truth and P is the set of top-k programs predicted by the model.

ference between Ansor's approach and existing approaches.

3 Design Overview

Ansor is an automated tensor program generation framework. Figure 4 shows the overall architecture of Ansor. The input of Ansor is a set of to be optimized DNNs. Ansor uses the operator fusion algorithm from Relay [42] to convert DNNs from popular model formats (*e.g.*, ONNX [6], TensorFlow PB) to partitioned small subgraphs. Ansor then generates tensor programs for these subgraphs. Ansor has three major components: (1) a program sampler that constructs a large search space and samples diverse programs from it; (2) a performance tuner that fine-tunes the performance of sampled programs; (3) a task scheduler that allocates time resources for optimizing multiple subgraphs in the DNNs.

Program sampler. One key challenge Ansor has to address is generating a large search space for a given computational graph. To cover diverse tensor programs with various high-level structures and low-level details, Ansor utilizes a hierarchical representation of the search space with two levels: *sketch* and *annotation* (§4). Ansor defines the high-level structures of programs as sketches and leaves billions of low-level choices (*e.g.*, tile size, parallel, unroll annotations) as annotations. This representation allows Ansor to enumerate high-level structures flexibly and sample low-level details efficiently. Ansor includes a program sampler that randomly samples programs from the space to provide comprehensive coverage of the search space.

Performance tuner. The performance of randomly sampled programs is not necessarily good. The next challenge is to fine-tune them. Ansor employs evolutionary search and a learned cost model to perform fine-tuning iteratively (§5). At each iteration, Ansor uses re-sampled new programs as well as good programs from previous iterations as the initial population to start the evolutionary search. Evolutionary search fine-tunes programs by mutation and crossover which perform out-of-order rewrite and address the limitation of sequential construction. Querying the learned cost model is orders of magnitude faster than actual measurement, so we can evaluate thousands of programs in seconds.

Task scheduler. Using program sampling and performance fine-tuning allows Ansor to find high-performance tensor programs for a computational graph. Intuitively, treating a whole DNN as a single computational graph and generating a full tensor program for it could potentially achieve the optimal performance. This, however, is inefficient because it has to deal with the unnecessary exponential explosion of the search space. Typically, the compiler partitions the large computational graph of a DNN into several small subgraphs [11, 42]. This partition has a negligible effect on the performance thanks to the layer-by-layer construction nature of DNNs. This brings the final challenge of Ansor: how to allocate time resources when generating programs for multiple subgraphs.

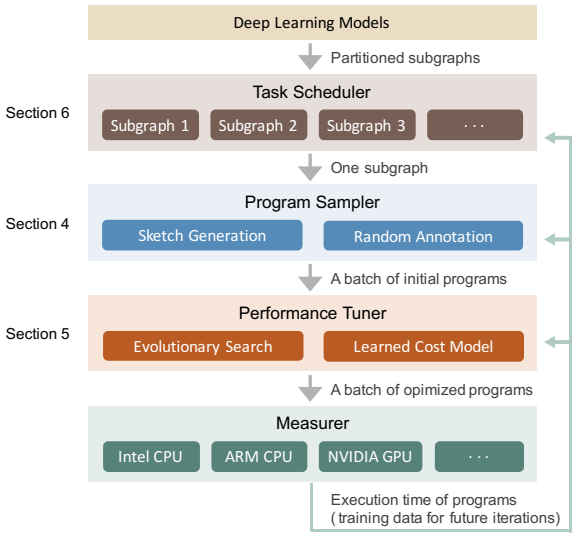


Figure 4: System Overview. The gray arrows show the flow of extracting subgraphs from deep learning models and generating optimized programs for them. The green arrows mean the measurer returns profiling data to update the status of all components in the system.

The task scheduler (§6) in Ansor uses a scheduling algorithm based on gradient descent to allocate resources to the subgraphs that are more likely to improve the end-to-end DNN performance.

4 Program Sampling

The search space an algorithm explores determines the best programs it can find. The considered search spaces in existing approaches are limited by the following factors: (1) Manual enumeration (*e.g.*, TVM [12]). It is impractical to manually enumerate all possible choices by templates, so existing manual templates only cover a limited search space heuristically. (2) Aggressive early pruning (*e.g.*, Halide auto-scheduler [2]). Aggressive early pruning based on evaluating incomplete programs prevents the search algorithm from exploring certain regions in the space.

In this section, we introduce techniques to push the boundary of the considered search space by addressing the above limitations. To solve (1), we automatically expand the search space by recursively applying a set of flexible derivation rules. To avoid (2), we randomly sample complete programs in the search space. Since random sampling gives an equal chance to every point to be sampled, our search algorithm can potentially explore every program in the considered space. We do not rely on random sampling to find the optimal program, because every sampled program is later fined-tuned (§5).

To sample programs that can cover a large search space, we define a hierarchical search space with two levels: *sketch* and *annotation*. We define the high-level structures of programs as sketches and leave billions of low-level choices (*e.g.*, tile

No	Rule Name	Condition	Application
1	Skip	$\neg IsStrictInlinable(S,i)$	$S' = S; i' = i-1$
2	Always Inline	$IsStrictInlinable(S,i)$	$S' = Inline(S,i); i' = i-1$
3	Multi-level Tiling	$HasDataReuse(S,i)$	$S' = MultiLevelTiling(S,i); i' = i-1$
4	Multi-level Tiling with Fusion	$HasDataReuse(S,i) \wedge HasFusibleConsumer(S,i)$	$S' = FuseConsumer(MultiLevelTiling(S,i),i); i' = i-1$
5	Add Cache Stage	$HasDataReuse(S,i) \wedge \neg HasFusibleConsumer(S,i)$	$S' = AddCacheWrite(S,i); i = i'$
6	Reduction Factorization	$HasMoreReductionParallel(S,i)$	$S' = AddRfactor(S,i); i' = i-1$
...	User Defined Rule	...	...

Table 1: Derivation rules used to generate sketches. The condition runs on the current state $\sigma = (S,i)$. The application derives the next state $\sigma' = (S',i')$ from the current state σ. Note that some function (*e.g.*, $AddRfactor$, $FuseConsumer$) can return multiple possible values of S'. In this case we collect all possible S', and return multiple next states σ' for a single input state σ.

size, parallel, unroll annotations) as annotations. At the top level, we generate sketches by recursively applying a few derivation rules. At the bottom level, we randomly annotate these sketches to get complete programs. This representation summarizes a few basic structures from billions of low-level choices, enabling the flexible enumeration of high-level structures and efficient sampling of low-level details.

While Ansor supports both CPU and GPU, we explain the sampling process for CPUs in §4.1 and §4.2 as an example. We then discuss how the process is different for GPU in §4.3.

4.1 Sketch Generation

As shown in Figure 4, the program sampler accepts partitioned subgraphs as input. The first column in Figure 5 shows two examples of the input. The input has three equivalent forms: the mathematical expression, the corresponding naive program obtained by directly expanding the loop indices, and the corresponding computational graph (directed acyclic graph, or DAG).

To generate sketches for a DAG with multiple nodes, we visit all the nodes in a topological order and build the structure iteratively. For computation nodes that are compute-intensive and have a lot of data reuse opportunities (*e.g.*, conv2d, matmul), we build basic tile and fusion structures for them as the sketch. For simple element-wise nodes (*e.g.*, ReLU, element-wise add), we can safely inline them. Note that new nodes (*e.g.*, caching nodes, layout transform nodes) may also be introduced to the DAG during the sketch generation.

We propose a derivation-based enumeration approach to generate all possible sketches by recursively applying several basic rules. This process takes a DAG as an input and returns a list of sketches. We define the State $\sigma = (S,i)$, where S is the current partially generated sketch for the DAG, and i is the index of the current working node. The nodes in a DAG are sorted in a topological order from output to input. The derivation begins from the initial naive program and the last node, or the initial state $\sigma = (naive\ program, index\ of\ the\ last\ node)$. Then we try to apply all derivation rules to the states recursively. For each rule, if the current state satisfies the application condition, we apply the rule to $\sigma = (S,i)$ and get $\sigma' = (S',i')$ where $i' \leq i$. This way the index i (working node) decreases monotonically. A state becomes a terminal state when $i = 0$. During enumeration, multiple rules can be applied to one state to generate multiple succeeding states. One rule can also generate multiple possible succeeding states. So we maintain a queue to store all intermediate states. The process ends when the queue is empty. All $\sigma.S$ in terminal states form a sketch list at the end of the sketch generation. The number of sketches is less than 10 for a typical subgraph.

Derivation rules. Table 1 lists derivation rules we used for the CPU. We first provide the definition of the used predicates and then describe the functionality of each rule. $IsStrictInliable(S,i)$ indicates if the node i in S is a simple element-wise operator that can always be inlined (*e.g.*, element-wise add, ReLU). $HasDataReuse(S,i)$ indicates if the node i in S is a compute-intensive operator and has plentiful intra-operator data reuse opportunity (*e.g.*, matmul, conv2d). $HasFusibleConsumer(S,i)$ indicates if the node i in S has only one consumer j and node j can be fused into node i (*e.g.*, matmul + bias_add, conv2d + relu). $HasMoreReductionParallel(S,i)$ indicates if the node i in S has little parallelism in space dimensions but has ample parallelism opportunity in reduction dimensions. (*e.g.*, computing 2-norm of a matrix, matmul $C_{2\times2} = A_{2\times512} \cdot B_{512\times2}$). We perform static analysis on the computation definitions to get the values for these predicates. The analysis is done automatically by parsing the read/write pattern in the mathematical expressions. Next, we introduce the functionality of each derivation rule.

Rule 1 just simply skips a node if it is not strictly inlinable. Rule 2 always inlines strictly inlinable nodes. Since the conditions of rule 1 and rule 2 are mutually exclusive, a state with $i > 1$ can always satisfy one of them and continue to derive.

Rules 3, 4, and 5 deal with the multi-level tiling and fusion for nodes that have data reuse. Rule 3 performs multi-level tiling for data reusable nodes. For CPU, we use a "SSRSRS" tile structure, where "S" stands for one tile level of space loops and "R" stands for one tile level of reduction loops. For example, in the matmul $C(i,j) = \sum_k A[i,k] \times B[k,j]$, i and j are space loops and k is a reduction loop. The "SSRSRS" tile structure for matmul expands the original 3-level loop (i,j,k) into a 10-level loop $(i_0,j_0,i_1,j_1,k_0,i_2,j_2,k_1,i_3,j_3)$. Although we do not permute the loop order, this multi-level

tiling can also cover some cases of reordering. For example, the above 10-level loop can be specialized to just a simple reorder (k_0, j_2, i_3) by setting the length of other loops to one. The "SSRSRS" tile structure is general for compute-intensive dense operators (*e.g.*, matmul, conv2d, conv3d) in deep learning, because they all consist of space loops and reduction loops.

Rule 4 performs multi-level tiling and also fuses the fusible consumers. For example, we fuse the element-wise nodes (*e.g.*, ReLU, bias add) into the tiled nodes (*e.g.*, conv2d, matmul). Rule 5 adds a caching node if the current data-reusable node does not have a fusible consumer. For example, the final output node in a DAG does not have any consumer, so it directly writes results into main memory by default and this is inefficient due to the high latency of memory accesses. By adding a cache node, we introduce a new fusible consumer into the DAG, then rule 4 can be applied to fuse this newly added cache node into the final output node. With the cache node fused, now the final output node writes its results into a cache block, and the cache block will be written to the main memory at once when all data in the block is computed.

Rule 6 can use `rfactor` [46] to factorize a reduction loop into a space loop to bring more parallelism.

Examples. Figure 5 shows three examples of the generated sketches. The sketches are different from the manual templates in TVM, because the manual templates specify both high-level structures and low-level details while sketches only define high-level structures. For the example input 1, the sorted order of the four nodes in the DAG is (A, B, C, D). To derive the sketches for the DAG, we start from output node $D(i=4)$ and apply rules to the nodes one by one. Specifically, the derivation for generated sketch 1 is:

$$\begin{aligned} \mathit{Input}\ 1 \rightarrow & \sigma(S_0, i=4) \xrightarrow{\text{Rule 1}} \sigma(S_1, i=3) \xrightarrow{\text{Rule 4}} \\ & \sigma(S_2, i=2) \xrightarrow{\text{Rule 1}} \sigma(S_3, i=1) \xrightarrow{\text{Rule 1}} \mathit{Sketch}\ 1 \end{aligned}$$

For the example input 2, the sorted order of the five nodes is (A, B, C, D, E). Similarly, we start from the output node $E(i=5)$ and apply rules recursively. The generated sketch 2 is derived by:

$$\begin{aligned} \mathit{Input}\ 2 \rightarrow & \sigma(S_0, i=5) \xrightarrow{\text{Rule 5}} \sigma(S_1, i=5) \xrightarrow{\text{Rule 4}} \\ & \sigma(S_2, i=4) \xrightarrow{\text{Rule 1}} \sigma(S_3, i=3) \xrightarrow{\text{Rule 1}} \\ & \sigma(S_4, i=2) \xrightarrow{\text{Rule 2}} \sigma(S_5, i=1) \xrightarrow{\text{Rule 1}} \mathit{Sketch}\ 2 \end{aligned}$$

Similarly, the generated sketch 3 is derived by:

$$\begin{aligned} \mathit{Input}\ 2 \rightarrow & \sigma(S_0, i=5) \xrightarrow{\text{Rule 6}} \sigma(S_1, i=4) \xrightarrow{\text{Rule 1}} \\ & \sigma(S_2, i=3) \xrightarrow{\text{Rule 1}} \sigma(S_3, i=2) \xrightarrow{\text{Rule 2}} \\ & \sigma(S_4, i=1) \xrightarrow{\text{Rule 1}} \mathit{Sketch}\ 3 \end{aligned}$$

Customization. While the presented rules are practical enough to cover the structures for most operators, there are always exceptions. For example, some special algorithms (*e.g.*, Winograd convolution [30]) and accelerator intrinsics (*e.g.*, TensorCore [37]) require special tile structures to be effective. Although the template-guided search approach (in TVM) can craft a new template for every new case, it needs a great amount of design effort. On the other hand, the derivation-based sketch generation in Ansor is flexible enough to generate the required structures for emerging algorithms and hardware, as we allow users to register new derivation rules and integrate them seamlessly with existing rules.

4.2 Random Annotation

The sketches generated by the previous subsection are incomplete programs because they only have tile structures without specific tile sizes and loop annotations, such as parallel, unroll, and vectorization. In this subsection, we annotate sketches to make them complete programs for fine-tuning and evaluation.

Given a list of generated sketches, we randomly pick one sketch, randomly fill out tile sizes, parallelize some outer loops, vectorize some inner loops, and unroll a few inner loops. We also randomly change the computation location of some nodes in the program to make a slight tweak to the tile structure. All "random" in this subsection means a uniform distribution over all valid values. If some special algorithms require custom annotations to be effective (*e.g.*, special unrolling), we allow users to give simple hints in the computation definition to adjust the annotation policy. Finally, since changing the layout of constant tensors can be done in compilation time and brings no runtime overhead, we rewrite the layouts of the constant tensors according to the multi-level tile structure to make them as cache-friendly as possible. This optimization is effective because the weight tensors of convolution or dense layers are constants for inference applications.

Examples of random sampling are shown in Figure 5. The sampled program might have fewer loops than the sketch because the loops with length one are simplified.

4.3 GPU Support

For GPU, we change the multi-level tiling structure from "SSRSRS" to "SSSRRSRS" to match the architecture of GPU. The loops in the first three space tiles are bound to BlockIdx, virtual thread (for reducing bank conflicts), and ThreadIdx, respectively. We add two sketch derivation rules, one for utilizing shared memory by inserting a caching node (similar to Rule 5) and the other for cross-thread reduction (similar to Rule 6).

5 Performance Fine-tuning

The programs sampled by the program sampler have good coverage of the search space, but their qualities are not guaranteed. This is because the optimization choices, such as tile structure and loop annotations, are all randomly sampled. In this

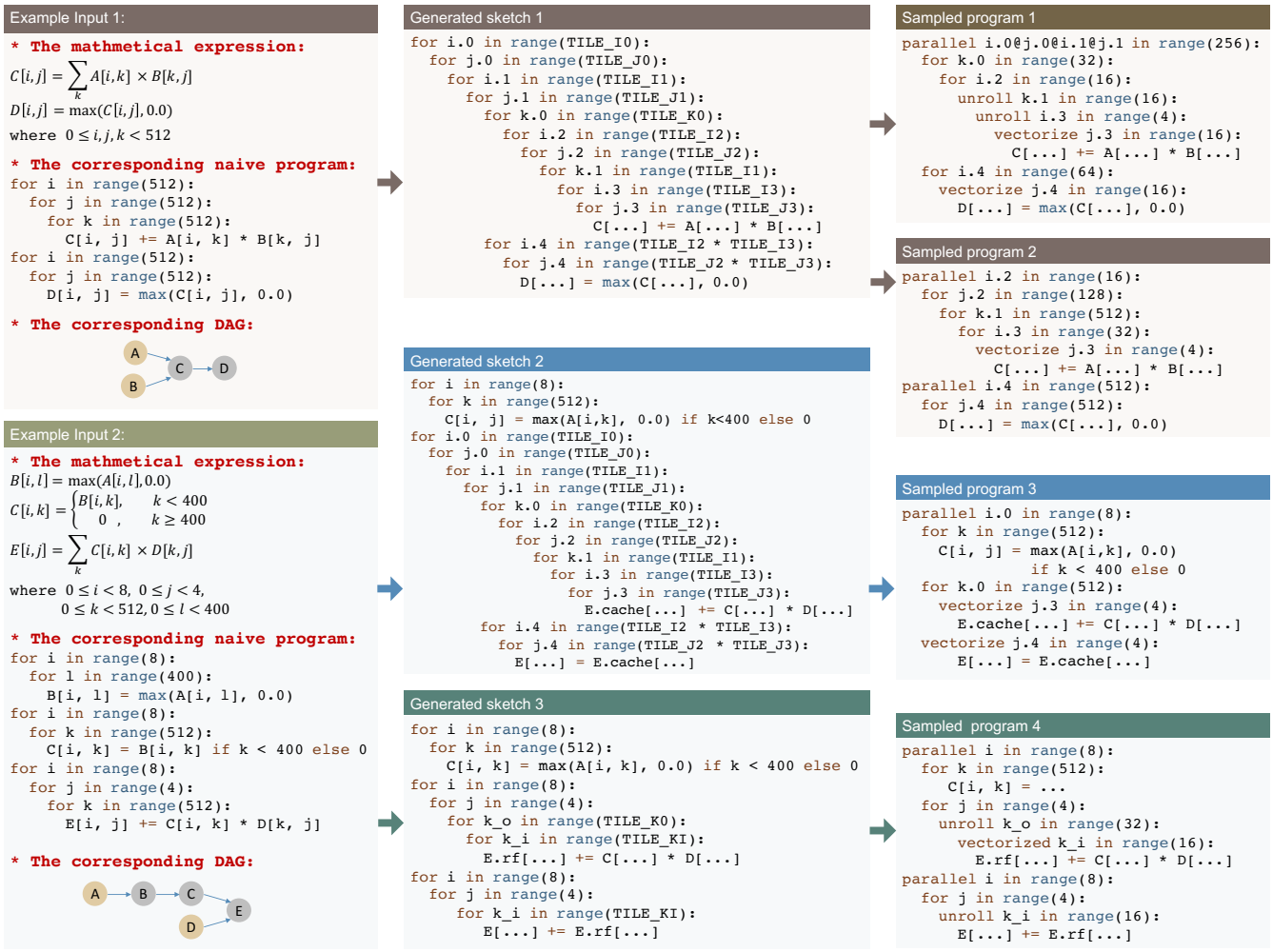


Figure 5: Examples of generated sketches and sampled programs. This figure shows two example inputs, three generated sketches and four sampled programs. The code example is pseudo code in a python-like syntax.

section, we introduce the performance tuner that fine-tunes the performance of the sampled programs via evolutionary search and a learned cost model.

The fine-tuning is performed iteratively. At each iteration, we first use evolutionary search to find a small batch of promising programs according to a learned cost model. We then measure these programs on hardware to get the actual execution time cost. Finally, the profiling data got from measurement is used to re-train the cost model to make it more accurate.

The evolutionary search uses randomly sampled programs as well as high-quality programs from the previous measurement as the initial population and applies mutation and crossover to generate the next generation. The learned cost model is used to predict the *fitness* of each program, which is the throughput of one program in our case. We run evolution for a fixed number of generations and pick the best programs found during the search. We leverage a learned cost model because the cost model can give relatively accurate estimations of the fitness of programs while being orders of magnitudes faster than the actual measurement. It allows us to compare tens of thousands of programs in the search space in seconds, and pick the promising ones to do actual measurements.

5.1 Evolutionary Search

Evolutionary search [54] is a generic meta-heuristic algorithm inspired by biological evolution. By iteratively mutating high-quality programs, we can generate new programs with potentially higher quality. The evolution starts from the sampled initial generation. To generate the next generation, we first select some programs from the current generation according to certain probabilities. The probability of selecting a program is proportional to its fitness predicted by the learned cost model (§5.2), meaning that the program with a higher performance score has a higher probability to be selected. For the selected programs, we randomly apply one of the evolution operations to generate a new program. Basically, for decisions we made during sampling (§4.2), we design corresponding evolution

operations to rewrite and fine-tune them.

Tile size mutation. This operation scans the program and randomly selects a tiled loop. For this tiled loop, it divides a tile size of one tile level by a random factor and multiplies this factor to another level. Since this operation keeps the product of tile sizes equal to the original loop length, the mutated program is always valid.

Parallel mutation. This operation scans the program and randomly selects a loop that has been annotated with parallel. For this loop, this operation changes the parallel granularity by either fusing its adjacent loop levels or splitting it by a factor.

Pragma mutation. Some optimizations in a program are specified by compiler-specific pragma. This operation scans the program and randomly selects a pragma. For this pragma, this operation randomly mutates it into another valid value. For example, our underlying code generator supports auto unrolling with a maximum number of steps by providing an `auto_unroll_max_step=N` pragma. We randomly tweak the number N.

Computation location mutation. This operation scans the program and randomly selects a flexible node that is not multi-level tiled (*e.g.*, a padding node in the convolution layer). For this node, the operation randomly changes its computation location to another valid attach point.

Node-based crossover. Crossover is an operation to generate new offspring by combining the genes from two or more parents. The genes of a program in Ansor are its rewriting steps. Every program generated by Ansor is rewritten from its initial naive implementation. Ansor preserves a complete rewriting history for each program during sketch generation and random annotation. We can treat rewriting steps as the genes of a program because they describe how this program is formed from the initial naive one. Based on this, we can generate a new program by combining the rewriting steps of two existing programs. However, arbitrarily combining rewriting steps from two programs might break the dependencies in steps and create an invalid program. As a result, the granularity of crossover operation in Ansor is based on nodes in the DAG, because the rewriting steps across different nodes usually have less dependency. Ansor randomly selects one parent for each node and merges the rewriting steps of selected nodes. When there are dependencies between nodes, Ansor tries to analyze and adjust the steps with simple heuristics. Ansor further verifies the merged programs to guarantee the functional correctness. The verification is simple because Ansor only uses a small set of loop transformation rewriting steps, and the underlying code generator can check the correctness by dependency analysis.

The evolutionary search leverages mutation and crossover to generate a new set of candidates repeatedly for several rounds and outputs a small set of programs with the highest scores. These programs will be compiled and measured on the target hardware to obtain the real running time cost. The collected measurement data is then used to update the cost model. In this way, the accuracy of the learned cost model is gradually improved to match the target hardware. Consequently, the evolutionary search gradually generates higher-quality programs for the target hardware platform.

Unlike the search algorithms in TVM and FlexTensor that can only work in a fixed grid-like parameter space, the evolutionary operations in Ansor are specifically designed for tensor programs. They can be applied to general tensor programs and can handle a search space with complicated dependency. Unlike the unfolding rules in Halide auto-scheduler, these operations can perform out-of-order modifications to programs, addressing the sequential limitations.

5.2 Learned Cost Model

A cost model is necessary for estimating the performance of programs quickly during the search. We adopt a learned cost model similar to related works [2, 12] with newly designed program features. A system based on learned cost models has great portability because a single model design can be reused for different hardware backends by feeding in different training data.

Since our target programs are mainly data parallel tensor programs, which are made by multiple interleaved loop nests with several assignment statements as the innermost statements, we train the cost model to predict the score of one innermost non-loop statement in a loop nest. For a full program, we make predictions for each innermost non-loop statement and add the predictions up as the score. We build the feature vector for an innermost non-loop statement by extracting features in the context of a full program. The extracted features include arithmetic features and memory access features. A detailed list of extracted features is in an appendix of the extended version of this paper [58].

We use weighted squared error as the loss function. Because we mostly care about identifying the well-performing programs from the search space, we put more weight on the programs that run faster. Specifically, the loss function of the model f on a program P with throughput y is $loss(f,P,y) = w_p(\sum_{s\in S(P)} f(s) - y)^2 = y(\sum_{s\in S(P)} f(s) - y)^2$ where $S(P)$ is the set of innermost non-loop statements in P. We directly use the throughput y as weight. We train a gradient boosting decision tree [9] as the underlying model f. A single model is trained for all tensor programs coming from all DAGs, and we normalize the throughput of all programs coming from the same DAG to be in the range of $[0,1]$. When optimizing a DNN, the number of measured programs are typically less than 30,000. Training a gradient boosting decision tree is very fast on such a small data sets, so we train a new model every time instead of doing incremental updates.

6 Task Scheduler

A DNN can be partitioned into many independent subgraphs (*e.g.*, conv2d + relu). For some subgraphs, spending time in tuning them does not improve the end-to-end DNN performance significantly. This is due to two reasons: either (1) the subgraph is not a performance bottleneck, or (2) tuning brings only minimal improvement in the subgraph's performance.

To avoid wasting time on tuning unimportant subgraphs, Ansor dynamically allocates different amounts of time resources to different subgraphs. Take ResNet-50 for example, it has 29 unique subgraphs after the graph partitioning. Most of these subgraphs are convolution layers with different shapes configurations (input size, kernel size, stride, etc). We need to generate different programs for different convolution layers because the best tensor program depends on these shape configurations. In reality, users may have multiple DNNs for all their applications. This leads to more subgraphs as well as more opportunities to reduce the total tuning time, because we can share and reuse knowledge between subgraphs. A subgraph can also appear multiple times in a DNN or across different DNNs.

We define a task as a process performed to generate high-performance programs for a subgraph. It means that optimizing a single DNN requires finishing dozens of tasks (*e.g.*, 29 tasks for ResNet-50). Ansor's task scheduler allocates time resources to tasks in an iterative manner. At each iteration, Ansor selects a task, generates a batch of promising programs for the subgraph, and measures the program on hardware. We define such an iteration as one unit of time resources. When we allocate one unit of time resources to a task, the task obtains an opportunity to generate and measure new programs, which also means the chance to find better programs. We next present the formulation of the scheduling problem and our solution.

6.1 Problem Formulation

When tuning a DNN or a set of DNNs, a user can have various types of goals, for example, reducing a DNN's latency, meeting latency requirements for a set of DNNs, or minimizing tuning time when tuning no longer improves DNN performance significantly. We thus provide users a set of objective functions to express their goals. Users can also provide their own objective functions.

Suppose there are n tasks in total. Let $t \in \mathbf{Z}^n$ be the allocation vector, where t_i is the number of time units spent on task i. Let the minimum subgraph latency task i achieves be a function of the allocation vector $g_i(t)$. Let the end-to-end cost of the DNNs be a function of the latency of the subgraphs $f(g_1(t), g_2(t), ..., g_3(t))$. Our objective is to minimize the end-to-end cost:

$$\text{minimize } f(g_1(t), g_2(t), ..., g_3(t))$$

$$f_1 = \sum_{j=1}^{m} \sum_{i \in S(j)} w_i \times g_i(t)$$
$$f_2 = \sum_{j=1}^{m} \max(\sum_{i \in S(j)} w_i \times g_i(t), L_j)$$
$$f_3 = -(\prod_{j=1}^{m} \frac{B_j}{\sum_{i \in S(j)} w_i \times g_i(t)})^{\frac{1}{m}}$$
$$f_4 = \sum_{j=1}^{m} \sum_{i \in S(j)} w_i \times \max(g_i(t), ES(g_i, t))$$

Table 2: Examples of objective functions for multiple neural networks

To minimize the end-to-end latency of a single DNN, we can define $f(g_1, g_2, ..., g_n) = \sum_{i=1}^{n} w_i \times g_i$, where w_i is the number of appearances of task i in the DNN. This formulation is straightforward because f is an approximation of the end-to-end DNN latency.

When tuning a set of DNNs, there are several options. Table 2 shows a number of example objective functions for tuning multiple DNNs. Let m be the number of DNNs, $S(j)$ is the set of tasks that belong to DNN j. f_1 adds up the latency of every DNN, which means to optimize the cost of a pipeline that sequentially runs all DNNs once. In f_2, we define L_j as the latency requirement of DNN j, meaning that we do not want to spend time on a DNN if its latency has already met the requirement. In f_3, we define B_j as the reference latency of a DNN j. As a result, our goal is to maximize the geometric mean of speedup against the given reference latency. Finally in f_4, we define a function $ES(g_i, t)$ that returns an early stopping value by looking at the history of latency of task i. It can achieve the effect of per-task early stopping.

6.2 Optimizing with Gradient Descent

We propose a scheduling algorithm based on gradient descent to efficiently optimize the objective function. Given the current allocation t, the idea is to approximate the gradient of the objective function $\frac{\partial f}{\partial t_i}$ in order to choose the task i such that $i = \text{argmax}_i |\frac{\partial f}{\partial t_i}|$. We approximate the gradient by making an optimistic guess and considering the similarity between tasks. The derivation is in an appendix of the extended version of this paper [58]. We approximate the gradient by

$$\frac{\partial f}{\partial t_i} \approx \frac{\partial f}{\partial g_i}(\alpha \frac{g_i(t_i) - g_i(t_i - \Delta t)}{\Delta t} + (1-\alpha)(\min(-\frac{g_i(t_i)}{t_i}, \beta \frac{C_i}{\max_{k \in N(i)} V_k} - g_i(t_i))))$$

where Δt is a small backward window size, $g_i(t_i)$ and $g_i(t_i - \Delta t)$ are known from the history of allocations. $N(i)$ is the set of similar tasks of i, C_i is the number of floating point operation in task i and V_k is the number of floating point operation per second we can achieve in task k. The parameter α and β control the weight to trust some predictions.

To run the algorithm, Ansor starts from $t = \mathbf{0}$ and warms up with a round of round-robin to get an initial allocation vector $t = (1, 1, ..., 1)$. After the warm-up, at each iteration, we

compute the gradient of each task and pick $\text{argmax}_i |\frac{\partial f}{\partial t_i}|$. Then we allocate the resource unit to task i and update the allocation vector $t_i = t_i + 1$. The optimization process continues until we run out of the time budget. To encourage exploration, we adopt a ε-greedy strategy [47], which preserves a probability of ε to randomly select a task.

Taking the case of optimizing for a single DNN's end-to-end latency for example, Ansor prioritizes a subgraph that has a high initial latency because our optimistic guess says we can reduce its latency quickly. Later, if Ansor spends many iterations on it without observing a decrease in its latency, Ansor leaves the subgraph because its $|\frac{\partial f}{\partial t_i}|$ decreases.

7 Evaluation

The core of Ansor is implemented in C++ with about 12K lines of code (3K for the search policy and 9K for other infrastructure). Ansor generates programs in its own intermediate representation (IR). These programs are then lowered to TVM IR for code generation targeting various hardware platforms. Ansor only utilizes TVM as a deterministic code generator.

We evaluate the performance of generated programs on three levels: single operator, subgraph, and entire neural network. For each level of evaluation, we compare Ansor against the state-of-the-art search frameworks and hardware-specific manual libraries. We also evaluate the search efficiency and the effectiveness of each component in Ansor.

The generated tensor programs are benchmarked on three hardware platforms: an Intel CPU (18-core Platinum 8124M@3.0 GHz), an NVIDIA GPU (V100), and an ARM CPU (4-core Cortex-A53@1.4GHz on the Raspberry Pi 3b+). We use float32 as the data type for all evaluations.

7.1 Single Operator Benchmark

Workloads. We first evaluate Ansor on a set of common deep learning operators, including 1D, 2D, and 3D convolution (C1D, C2D, and C3D respectively), matrix multiplication (GMM), group convolution (GRP), dilated convolution (DIL) [57], depth-wise convolution (DEP) [24], transposed 2D convolution (T2D) [40], capsule 2D convolution (CAP) [23], and matrix 2-norm (NRM). For each operator, we select 4 common shape configurations and evaluate them with two batch sizes (1 and 16). In total, there are 10 operators $\times 4$ shape configurations $\times 2$ batch size $= 80$ test cases. The shape configurations used can be found in an appendix of the extended version of this paper [58]. We run these test cases on the Intel CPU.

Baselines. We include PyTorch (v1.5) [39], Halide auto-scheduler (commit: 1f875b0) [2], FlexTensor (commit: 7ac302c) [59], and AutoTVM (commit: 69313a7) [12] as baselines. PyTorch is backed by the vendor-provided kernel library MKL-DNN [27]. Halide auto-scheduler is a sequential construction based search framework for Halide. AutoTVM and FlexTensor are template-guided search frameworks based on TVM. Since Halide auto-scheduler does not have a pre-trained cost model for AVX-512, we disabled AVX-512 for the evaluation in §7.1 and §7.2. For every operator, we use the best layout available in each framework, but the input and output tensors must not be packed.

Search settings. We let search frameworks (*i.e.*, Halide auto-scheduler, FlexTensor, AutoTVM, and Ansor) to run search or auto-tuning with up to $1,000$ measurement trials per test case. This means each framework can measure at most $80 \times 1,000$ programs for auto-tuning in this evaluation. Using the same number of measurement trials makes it a fair comparison without involving implementation details. In addition, using $1,000$ measurement trials per test case is typically enough for the search to converge in these frameworks.

Normalization. Figure 6 shows the normalized performance. For each test case, we normalize the throughputs to the best performing framework. We then plot the geometric mean of the four shapes of each operator. The geometric mean is also normalized to the best performing framework, so the best framework has a normalized performance of 1 in the figure. The error bar denotes the standard deviation of the normalized throughput of four shapes of each operator.

Results. As shown in the Figure 6, Ansor performs the best or equally the best in all operator and batch size settings. Ansor outperforms existing search frameworks by $1.1 - 22.5\times$. The performance improvements of Ansor come from both its large search space and effective exploration strategy. For most operators, we found the best program generated by Ansor is outside the search space of existing search frameworks because Ansor is able to explore more optimization combinations. For example, the significant speedup on NRM is because Ansor can parallelize reduction loops, while other frameworks do not. The large speedup on T2D is because Ansor can use correct tile structures and unrolling strategies to let the code generator simplify the multiplication of zeros in strided transposed convolution. In contrast, other frameworks fail to capture many effective optimizations in their search space, making them unable to find the programs that Ansor does. For example, the unfolding rules in Halide do not split the reduction loop in GMM and do not split reduction loops in C2D when padding is computed outside of reduction loops. The templates in AutoTVM have limited tile structures, as they cannot cover the structure of "Generated Sketch 1" in Figure 5. The template in FlexTensor does not change the computation location of padding. The template in FlexTensor fails to run for reduction operators like NRM.

Ablation study. We run four variants of Ansor on a convolution operator and report the performance curve. We pick the last convolution operator in ResNet-50 with batch size=16 as the test case, because its search space is sufficiently large to evaluate the search algorithms. Other operators share a similar pattern. In Figure 7, each curve is the median of 5 runs. "Ansor (ours)" uses all our introduced techniques. "Beam

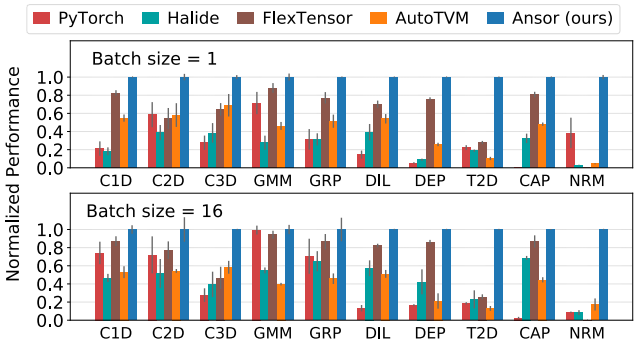


Figure 6: Single operator performance benchmark on a 20-core Intel-Platinum-8269CY. The y-axis is the throughput normalized to the best throughput for each operator.

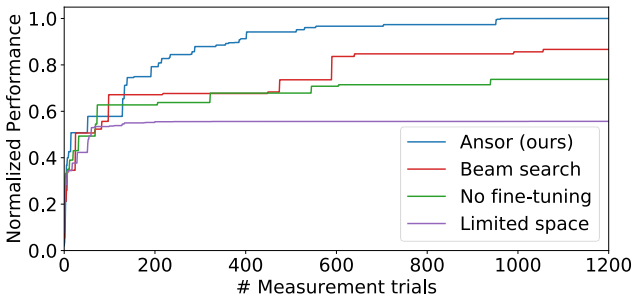


Figure 7: Ablation study of four variants of Ansor on a convolution operator. The y-axis is the throughput relative to the throughput of the best program.

Search" means we prune incomplete programs with the cost model during the sampling process and do not use fine-tuning. "No fine-tuning" is based on "Ansor (ours)" but disables fine-tuning and only relies on random sampling. "Limited space" is also based on "Ansor (ours)" but limits the search space to make it similar to the space in existing manual templates (*e.g.*, limit tiling level, innermost tile sizes, and computation location). As demonstrated by Figure 7, dropping either the large search space or efficient fine-tuning decreases the final performance significantly. The aggressive early pruning in "Beam search" throws away incomplete programs with good final performance due to inaccurate estimation.

7.2 Subgraph Benchmark

We perform the subgraph benchmark on two common subgraphs in DNNs. The "ConvLayer" is a subgraph consisting of 2D convolution, batch normalization [28], and ReLU activation, which is a common pattern in convolutional neural networks. The "TBS" is a subgraph consisting of two matrix transposes, one batch matrix multiplication, and a softmax, which is a pattern in the multi-head attention [51] in language models. Similar to the single operator benchmark (§7.1), we select four different shape configurations and two batch sizes, run auto-tuning with up to 1,000 measurement trails per test case, and report the normalized performance. We use the same set of baseline frameworks and run the benchmark on the Intel CPU and the NVIDIA GPU. We do not report the performance of Halide auto-scheduler on GPU because as of writing the paper its GPU support is still in an experimental stage. FlexTensor fails to run on complicated subgraphs like "TBS".

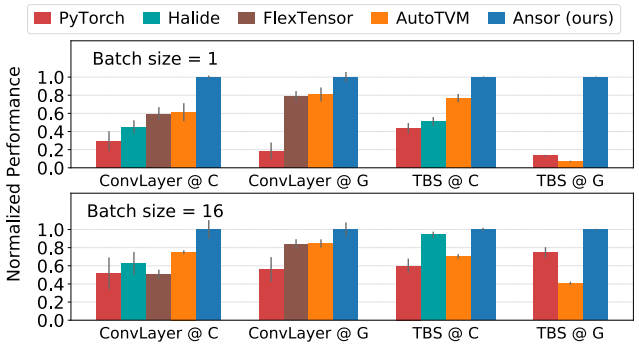


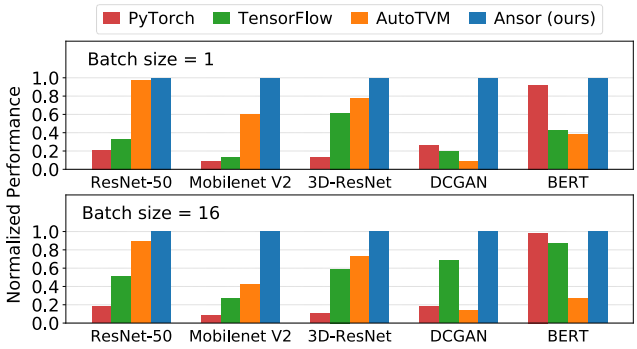
Figure 8: Subgraph performance benchmark on a 20-core Intel-Platinum-8269CY and an NVIDIA V100. "@C" denotes CPU results and "@G" denotes GPU results. The y-axis is the throughput normalized to the best throughput for each subgraph.

Figure 8 shows that Ansor outperforms manual libraries and other search frameworks by $1.1-14.2\times$. Ansor can generate high-performance programs consistently for these subgraphs on both platforms. FlexTensor performs well for single operators but shows less advantage for subgraphs because it lacks the support of operator fusion.

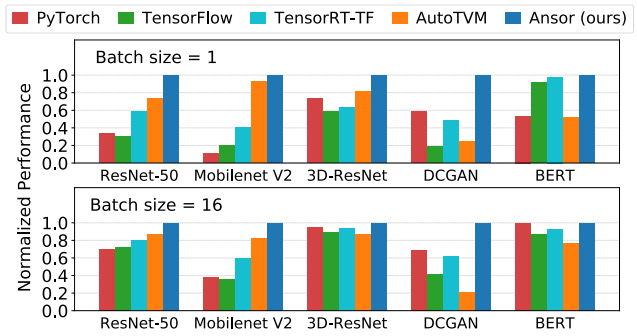
7.3 End-to-End Network Benchmark

Workloads. We benchmark the end-to-end inference execution time of several DNNs, which include ResNet-50 [22] and MobileNet-V2 [43] for image classification, 3D-ResNet-18 [21] for action recognition, DCGAN [40] generator for image generation, and BERT [15] for language understanding. We benchmark these DNNs on three hardware platforms. For the server-class Intel CPU and NVIDIA GPU, we report the results for batch size 1 and batch size 16. For the ARM CPU in the edge device, real-time feedback is typically desired, so we only report the results for batch size 1.

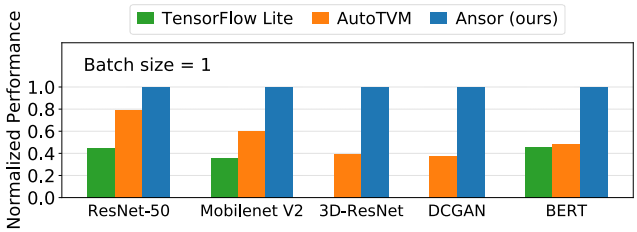
Baselines and Settings. We include PyTorch (v1.5 with torch script), TensorFlow (v2.0 with graph mode), TensorRT (v6.0 with TensorFlow integration) [38], TensorFlow Lite (V2.0), and AutoTVM as baseline frameworks. We do not include Halide auto-scheduler or FlexTensor because they lack the support of widely-used deep learning model formats (*e.g.*, ONNX, TensorFlow PB) and high-level graph optimizations. As a result, we expect that the end-to-end execution time they can achieve will be the sum of the latency of all subgraphs in a DNN. In contract, AutoTVM can optimize a whole DNN with its manual templates and various graph-level optimizations (*e.g.*, graph-level layout search [32], graph-level constant



(a) Intel CPU



(b) NVIDIA GPU



(c) ARM CPU

Figure 9: Network inference performance benchmark on three hardware platforms. The y-axis is the throughput relative to the best throughput for each network.

folding [42]) which improve the performance significantly. Ansor also performs layout rewrite as described in §4.2. We let both AutoTVM and Ansor run auto-tuning until they use to $1000 \times n$ measurement trials on each DNN, where n is the number of subgraphs in the DNN. This is typically enough for them to converge. We set the objective of the task scheduler as minimizing the total latency of one DNN and generate programs for these networks one by one. On the other hand, PyTorch, TensorFlow, TensorRT, and TensorFlow Lite are all backed by static kernel libraries (MKL-DNN on Intel CPU, CuDNN on NVIDIA GPU, and Eigen on ARM CPU) and do not need auto-tuning. We enable AVX-512 for all frameworks on the Intel CPU in this network benchmark.

Results. Figure 9 shows the results on the Intel CPU, NVIDIA GPU and ARM CPU [2]. Overall, Ansor performs the best or equally the best in all cases. Compared with search-based AutoTVM, Ansor matches or outperforms it in all cases with $1.0-21.8\times$ speedup. Compared with the best alternative, Ansor improves the execution performance of DNNs on the Intel CPU, ARM CPU, and NVIDIA GPU by up to $3.8\times$, $2.6\times$, and $1.7\times$, respectively. The reason for the significant speedup on DCGAN is that DCGAN mainly consists of transposed 2D convolution (T2D), which can be well optimized by Ansor, as shown and explained in the single operator benchmark (§7.1). AutoTVM performs very well for ResNet-50 on the Intel CPU thanks to its highly-optimized templates for 2D convolution and global layout search [32]. Ansor does not run a global layout search but does rewrite the layout of weight tensors as described in §4.2. Ansor uses more levels of tiling so it packs weight tensors into more levels. The layout rewrite brings about 40% improvement to ResNet-50 in Ansor. Compared with vendor-specific static libraries, Ansor has more advantages on uncommon shapes and small batch sizes, because it is not easy to manually optimize for these cases.

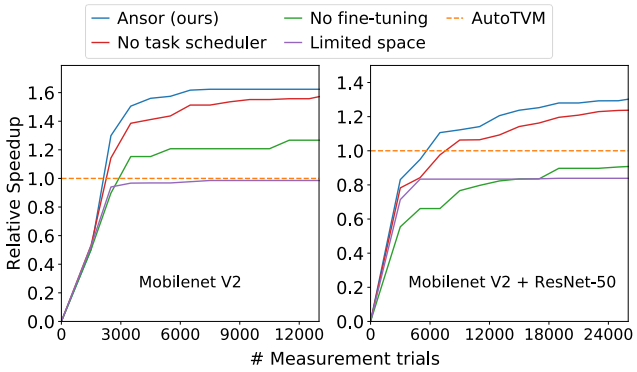


Figure 10: Network performance auto-tuning curve. The y-axis is the speedup relative to AutoTVM.

Ablation study. We run variants of Ansor on two test cases in Figure 10. In the left figure, we run four variants of Ansor to generate programs for a single mobilenet-V2. In the right figure, we run these variants for both mobilenet-V2 and ResNet-50. We set the objective function of the task scheduler to be the geometric mean of speedup against AutoTVM. As shown in Figure 10, "No task scheduler" means we use a round-robin strategy to allocate equal time resources to all subgraphs. "Limited space" is based on "Ansor (ours)" but limits the search space. "No fine-tuning" is also based on "Ansor (ours)" but disables fine-tuning and relies on random sampling only. As can be seen in Figure 10, "Limited space" performs the worst in terms of the final achieved performance, proving that the best programs are not included in the limited space. The final achieved performance can be improved by enlarging the search space, as depicted in "No fine-tuning". However, in the right figure, randomly assigning tile sizes and annotations

[2] 3D-ResNet and DCGAN are not yet supported by TensorFlow Lite on the ARM CPU.

still cannot beat AutoTVM in the given time budget. After enabling fine-tuning, "No task scheduler" outperforms AutoTVM in both cases. Finally, "Ansor (ours)" employs the task scheduler to prioritize performance bottlenecks (*e.g.*, subgraphs contain 3x3 convolution), so it performs the best in both search efficiency and the final achieved performance.

7.4 Search Time

Ansor searches efficiently and can outperform or match AutoTVM with less search time. Ansor slices the time and utilizes the task scheduler to simultaneously optimize all subgraphs together. In contrast, AutoTVM and other systems do not have a task scheduler, so they generate programs for all subgraphs one by one with a predefined budget of measurement trials for each subgraph. Ansor saves the search time by prioritizing important subgraphs, while AutoTVM spends predefined time budget on every subgraph, which may be a waste on the unimportant subgraphs.

Table 3 shows the search time required for Ansor to match the performance of AutoTVM on the Intel CPU network benchmark (§7.3). We list the search time in two metrics: number of measurements and wall-clock time. "Number of measurements" is a metric agnostic to the implementation of measurement and the overhead of search algorithm, while "Wall-clock time" takes these factors into account. As shown in the table, Ansor can match the performance of AutoTVM with an order of magnitude less search time. In Table 3a the saving in search time comes from the task scheduler, efficient fine-tuning, and comprehensive coverage of effective optimizations. In Table 3b, Ansor shows more time-saving in wall-clock time. This is because Ansor does not introduce much search overhead and has a better implementation of the measurement (on the Intel CPU, Ansor can get accurate measurement results with fewer repetitions by explicitly flushing the cache for some tensors). On other backends, Ansor can match the performance of AutoTVM with a similar saving in search time.

Typically, it takes several hours for Ansor to generate fully-optimized programs for a DNN on a single machine. This is acceptable for inference applications because it is a one-shot effort before deployment. In addition, the whole architecture of Ansor can be parallelized very easily.

7.5 Cost Model Evaluation

In this subsection, we evaluate the prediction quality of the learned cost model. We use 25,000 programs measured during tuning ResNet-50 on the Intel CPU as the data set. We randomly pick 20,000 programs as the training set and use the remaining 5,000 programs as the test set. We train the cost model and let it make predictions for the test set.

Figure 11 plots the predicted throughputs vs. measured throughputs. The measured throughputs are normalized to

	AutoTVM	Ansor	Time-saving
ResNet-50	21,220	6,403	3.3 ×
Mobilenet-V2	31,272	1,892	16.5 ×
3D-ResNet	5,158	1,927	2.7 ×
DCGAN	3,003	298	10.1 ×
BERT	6,220	496	12.5 ×

(a) The number of measurements.

	AutoTVM	Ansor	Time-saving
ResNet-50	39,250	4,540	8.6 ×
Mobilenet-V2	58,468	660	88.6 ×
3D-ResNet	7,594	2,296	3.3 ×
DCGAN	4,914	420	11.7 ×
BERT	12,007	266	45.1 ×

(b) Wall-clock time (seconds)

Table 3: The number of measurements and wall-clock time used for Ansor to match the performance of AutoTVM on the Intel CPU (batch size=1).

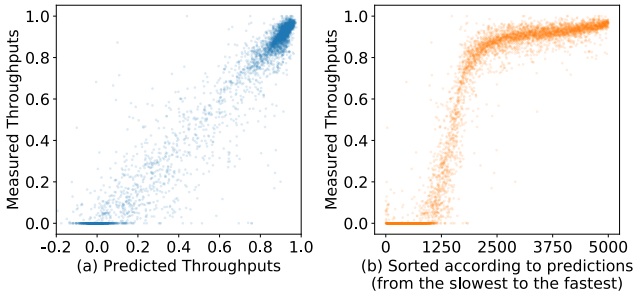


Figure 11: Measured throughputs vs. predicted throughputs.

the best performing programs in the test set. The predicted throughputs are the output of the model, so they can be negative. In Figure 11a, the points scatter around the diagonal line, meaning that the model makes accurate predictions. The distribution is not uniform because the data set is collected during the search. Good programs have a higher probability to be chosen for measurements, so most of the programs are in the top right corner. The points with measured throughput 0.0 are programs that are invalid or killed due to timeout during measurements. In Figure 11b, we sort the 5000 points according to the predictions from the slowest to the fastest, and use the relative ranking as x-axis. So the points are distributed uniformly over x-axis. It shows the distribution of performance of the explored programs better.

The model archives 0.079 RMSE, 0.958 R^2 correlation, 0.851 pairwise comparison accuracy, and 0.624 recall@30 of top-30 programs (see the definition at footnote 1) on the test set.

8 Related Work

Automatic tensor program generation based on scheduling languages. Halide [41] introduces a scheduling language

that can describe loop optimization primitives. This language is suitable for both manual optimization and automatic search. Halide has three versions of auto-scheduler based on different techniques [2, 31, 36]. The latest one with beam search and learned cost model performs the best among them, which is also used in our evaluation. TVM [11] utilizes a similar scheduling language and includes a template-guided search framework AutoTVM [12]. FlexTensor [59] proposes general templates that can target a set of operators, but its templates are designed for single operators. It is hard to use these templates for optimizations involving multiple operators (e.g., operator fusion). A concurrent work ProTuner [19] uses Monte Carlo tree search to solve the inaccurate estimation problem in Halide auto-scheduler. ProTuner mainly targets image processing workloads, while Ansor targets deep learning workloads and introduces new search space and other optimizations.

Polyhedral compilation models. The polyhedral compilation model [8,52,53] formulates the optimization of programs as an integer linear programming (ILP) problem. It optimizes a program with affine loop transformation that minimizes the data reuse distance between dependent statements. Tiramisu [5] and TensorComprehensions [49] are two polyhedral compilers that also target the deep learning domain. Tiramisu provides a scheduling language similar to the Halide language, and it needs manual scheduling. TensorComprehensions can search for GPU code automatically, but it is not yet meant to be used for compute-bounded problems [11]. It cannot outperform TVM on operators like conv2d and matmul [11,48]. This is because of the lack of certain optimizations [50] and the inaccurate implicit cost model in the polyhedral formulation.

Graph-level optimization for deep learning. Graph-level optimizations treat an operator in the computational graph as a basic unit and perform optimization at graph level without changing the internal implementations of operators. The common optimizations at graph level include layout optimizations [32], operator fusion [11, 38, 60], constant folding [42], auto-batching [33], automatic generation of graph substitution [29] and so forth. The graph-level optimizations are typically complementary to operator-level optimizations. Graph-level optimizations can also benefit from high-performance implementations of operators. For example, general operator fusion relies on the code generation ability of Ansor. We leave the joint optimization of Ansor and more graph-level optimization as future work.

Search-based compilation and auto-tuning. Search based compilation and auto-tuning have already shown their effectiveness in domains other than deep learning. Stock [44] is a super-optimizer based on random search. Stock searches for loop-free hardware instruction sequences, while Ansor generates tensor programs with nests of loops. OpenTuner [4] is a general framework for program auto-tuning based on multi-armed bandit approaches. OpenTuner relies on user-specified search space, while Ansor constructs the search space automatically. Traditional high-performance libraries such as ATLAS [56] and FFTW [16] also utilize auto-tuning. More recent works NeuroVectorizer [18] and AutoPhase [20, 26] use deep reinforcement learning to automatically vectorize programs and optimize the compiler phase ordering.

9 Limitations and Future work

One of Ansor's limitations is that Ansor cannot optimize graphs with dynamic shapes [45]. Ansor requires the shapes in the computational graph to be static and known in advance to do analysis, construct the search space, and perform measurements. How to generate programs for symbolic or dynamic shape is an interesting future direction. Another limitation is that Ansor only supports dense operators. To support sparse operators (*e.g.*, SpMM) that are commonly used in sparse neural networks [17] and graph neural networks [25], we expect that a large portion of Ansor can still be reused, but we need to redesign the search space. Lastly, Ansor only performs program optimizations at a high level but relies on other code generators (*e.g.*, LLVM and NVCC) to do platform-dependent optimizations (*e.g.*, instruction selection). Ansor comes short of utilizing the special instructions, such as Intel VNNI, NVIDIA Tensor Core, and ARM Dot for mixed-precision and low-precision operators, which are not handled well by the off-the-shelf code generators currently.

10 Conclusion

We propose Ansor, an automated search framework that generates high-performance tensor programs for deep neural networks. By efficiently exploring a large search space and prioritizing performance bottlenecks, Ansor finds high-performance programs that are outside the search space of existing approaches. Ansor outperforms existing manual libraries and search-based frameworks on a diverse set of neural networks and hardware platforms by up to 3.8×. By automatically searching for better programs, we hope that Ansor will help bridge the gap between the increasing demand in computing power and limited hardware performance. Ansor is integrated into the Apache TVM open-source project [3].

11 Acknowledgement

We would like to thank Weizhao Xian, Tianqi Chen, Frank Luan, anonymous reviewers, and our shepherd, Derek Murray, for their insightful feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

[3] https://tvm.apache.org/

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[3] Hassan Abu Alhaija, Siva Karthik Mustikovela, Lars Mescheder, Andreas Geiger, and Carsten Rother. Augmented reality meets deep learning for car instance segmentation in urban scenes. In *British machine vision conference*, volume 1, page 2, 2017.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: an extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: a polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[6] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: open neural network exchange, 2019.

[7] Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019.

[8] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[9] Tianqi Chen and Carlos Guestrin. Xgboost: a scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[12] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[14] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] Matteo Frigo and Steven G Johnson. Fftw: an adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pages 1381–1384. IEEE, 1998.

[17] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[18] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: end-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255, 2020.

[19] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanović, and Ion Stoica. Protuner: tuning programs with monte carlo tree search. *arXiv preprint arXiv:2005.13685*, 2020.

[20] Ameer Haj-Ali, Qijing Huang, William Moses, John Xiang, John Wawrzynek, Krste Asanovic, and Ion Stoica. Autophase: juggling hls phase orderings in random forests with deep reinforcement learning. In *Third Conference on Machine Learning and Systems (ML-Sys)*, 2020.

[21] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Can spatiotemporal 3d cnns retrace the history of 2d cnns and imagenet? In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6546–6555, 2018.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with em routing. 2018.

[24] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[25] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. Featgraph: A flexible and efficient backend for graph neural network systems. *arXiv preprint arXiv:2008.11359*, 2020.

[26] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308. IEEE, 2019.

[27] Intel. Intel® math kernel library for deep learning networks, 2017.

[28] Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[29] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[30] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[31] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG)*, 37(4):139, 2018.

[32] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing cnn model inference on cpus. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, 2019.

[33] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[34] Mark F. Medress, Franklin S Cooper, Jim W. Forgie, CC Green, Dennis H. Klatt, Michael H. O'Malley, Edward P Neuburg, Allen Newell, DR Reddy, B Ritea, et al. Speech understanding systems: report of a steering committee. *Artificial Intelligence*, 9(3):307–316, 1977.

[35] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[36] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[37] Nvidia. Nvidia tensor cores, 2017.

[38] Nvidia. Nvidia tensorrt: programmable inference accelerator, 2017.

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[40] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image

processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[42] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: a high-level compiler for deep learning. *arXiv preprint arXiv:1904.08368*, 2019.

[43] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[44] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[45] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *arXiv preprint arXiv:2006.03031*, 2020.

[46] Patricia Suriana, Andrew Adams, and Shoaib Kamil. Parallel associative reductions in halide. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 281–291. IEEE, 2017.

[47] Richard S Sutton and Andrew G Barto. *Reinforcement learning: an introduction*. MIT press, 2018.

[48] Philippe Tillet, HT Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[49] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[50] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. The next 700 accelerated layers: from mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–26, 2019.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[52] Sven Verdoolaege. Presburger formulas and polyhedral compilation. 2016.

[53] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.

[54] Pradnya A Vikhar. Evolutionary algorithms: a critical review and its future prospects. In *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*, pages 261–265. IEEE, 2016.

[55] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. A unified optimization approach for cnn model inference on integrated gpus. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[56] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 38–38. IEEE, 1998.

[57] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[58] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: generating high-performance tensor programs for deep learning. https://arxiv.org/abs/2006.06762, 2020.

[59] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: an automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[60] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. Fusionstitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924*, 2020.



RAMMER: Enabling Holistic Deep Learning Compiler Optimizations with *r*Tasks

Lingxiao Ma[*] [†◇] Zhiqiang Xie[*] [‡◇] Zhi Yang[†] Jilong Xue[◇] Youshan Miao[◇]
Wei Cui[◇] Wenxiang Hu[◇] Fan Yang[◇] Lintao Zhang[◇] Lidong Zhou[◇]

[†]*Peking University* [‡]*ShanghaiTech University* [◇]*Microsoft Research*

Abstract

Performing Deep Neural Network (DNN) computation on hardware accelerators efficiently is challenging. Existing DNN frameworks and compilers often treat the DNN operators in a data flow graph (DFG) as opaque library functions and schedule them onto accelerators to be executed individually. They rely on another layer of scheduler, often implemented in hardware, to exploit the parallelism available in the operators. Such a two-layered approach incurs significant scheduling overhead and often cannot fully utilize the available hardware resources. In this paper, we propose RAMMER, a DNN compiler design that optimizes the execution of DNN workloads on massively parallel accelerators. RAMMER generates an efficient static spatio-temporal schedule for a DNN at compile time to minimize scheduling overhead. It maximizes hardware utilization by holistically exploiting parallelism through inter- and intra- operator co-scheduling. RAMMER achieves this by proposing several novel, hardware neutral, and clean abstractions for the computation tasks and the hardware accelerators. These abstractions expose a much richer scheduling space to RAMMER, which employs several heuristics to explore this space and finds efficient schedules. We implement RAMMER for multiple hardware backends such as NVIDIA GPUs, AMD GPUs, and Graphcore IPU. Experiments show RAMMER significantly outperforms state-of-the-art compilers such as TensorFlow XLA and TVM by up to 20.1×. It also outperforms TensorRT, a vendor optimized proprietary DNN inference library from NVIDIA, by up to 3.1×.

1 Introduction

Deep neural network (DNN) is now a widely adopted approach for image classification, natural language processing, and many other AI tasks. Due to its importance, many computational devices, such as CPU, GPU, FPGA, and specially designed DNN accelerators have been leveraged to perform DNN computation. Efficient DNN computation on these devices is an important topic that has attracted much research attention in recent years [23, 28, 32, 40, 52]. One of the key factors that affect the efficiency of DNN computation is scheduling, i.e. deciding the order to perform various pieces of computation on the target hardware. The importance of scheduling in general is well known and has been thoroughly studied [20, 39]. However, there is little work discussing scheduling for DNN computation on hardware devices specifically.

The computational pattern of a deep neural network is usually modeled as a data flow graph (DFG), where each node corresponds to an operator, which represents a unit of computation such as matrix multiplication, while an edge depicts the dependency between operators. This representation naturally contains two levels of parallelism. The first level is the inter-operator parallelism, where operators that do not have dependencies in the DFG may run in parallel. The second level is the intra-operator parallelism, where an operator such as matrix multiplication has inherent internal data parallelism and can leverage hardware accelerators that can perform parallel computation, such as a GPU.

To exploit the two levels of parallelism, current practice adopts a two-layered scheduling approach. An inter-operator DFG layer scheduler takes the data flow graph and emits operators that are ready to be executed based on the dependencies. In addition, an intra-operator scheduler takes an operator and maps it to the parallel execution units in the accelerator. This layering design has a fundamental impact on the system architectures of the existing DNN tool sets. For example, the DFG layer scheduler is typically implemented in deep learning frameworks such as TensorFlow [18] or ONNX Runtime [14]. The operator layer scheduler, on the other hand, is often hidden behind the operator libraries such as cuDNN [12] and MKL-DNN [9], and sometimes implemented directly in hardware, as is the case for GPUs.

While widely adopted by existing frameworks and accelerators, such a two-layer scheduling approach incurs fundamental performance limitations. The approach works well only

[*]Both authors contributed equally.

when the overhead of emitting operators is largely negligible compared to the execution time of operators, and when there is sufficient intra-operator parallelism to saturate all processing units in an accelerator. This unfortunately is often not the case in practice. DNN accelerators keep on increasing performance at a much faster pace than CPUs, thus making the operator emitting overhead more and more pronounced. This is exacerbated for DNN inference workloads when the batch size is small, which limits the intra-operator parallelism. Moreover, the two-layer scheduling approach overlooks the subtle interplay between the upper and lower layers: to optimize the overall performance, a system could reduce the degree of intra-operator parallelism in order to increase the level of inter-operator parallelism (§ 2).

To mitigate these limitations, we present RAMMER, a deep learning compiler that takes a holistic approach to manage the parallelism available in the DNN computation for scheduling. It unifies the inter- and intra-operator scheduling through a novel abstraction called *r*Task. *r*Task enables the scheduler to break the operator boundary and allows fine-grained scheduling of computation onto devices. Instead of the existing design that breaks scheduling into two pieces managed by software and hardware separately, RAMMER is a unified software-only solution, which makes it less dependent on underlying hardware and thus can be adopted by diverse DNN accelerators. In RAMMER, we make the following design decisions.

First, to exploit the intra-operator parallelism through a software compiler, RAMMER redefines a DNN operator as an *rTask*-operator or *r*Operator. An *r*Operator consists of multiple independent, homogeneous *rTask*s, each is a minimum schedulable unit runs on a single execution unit of an accelerator (e.g., a streaming multiprocessor SM in a GPU). Thus, *r*Task as the fine-grained intra-operator information is exposed to the RAMMER scheduler. RAMMER treats a DNN as a data flow graph of *r*Operator nodes, hence it can still see the coarse-grained inter-operator (DFG) dependencies.

Unfortunately, certain modern accelerators such as GPU do not expose interfaces for intra-operator (i.e., *r*Task) scheduling. To address this challenge, as a second design decision RAMMER abstracts a hardware accelerator as a *virtualized parallel device* (vDevice), which contains multiple virtualized execution units (vEU). The vDevice allows several *r*Tasks, even from different operators, to run on a specified vEU in a desired order. Moreover, a vEU can run a *barrier* *r*Task that waits for the completion of a specified set of *r*Tasks, thus ensuring the correct execution of *r*Tasks from dependent operators. The vDevice maps a vEU to one of the physical execution units in an accelerator to perform the actual computation of *r*Tasks.

Finally, fine-grained scheduling could incur significant runtime overheads, even more so than the operator scheduling overhead discussed previously. To address this issue, RAMMER moves the scheduling decision from runtime to compile time. This is driven by the observation that most DNN's DFG is available at the compile time, and the operators usually exhibit deterministic performance characteristics. Therefore, the runtime performance can be obtained through compile time profiling [45]. This not only avoids unnecessary runtime overheads, but also allows a more costly scheduling policy to fully exploit the inter- and intra- operator parallelism together.

RAMMER is compatible with optimizations developed in existing DNN compilers. RAMMER can import a data-flow graph from other frameworks like TensorFlow. Such a DFG can be optimized with techniques employed by a traditional graph optimizer such as [18]. An *r*Operator can also be optimized by an existing kernel tuner [23]. Our experience shows that, on top of existing optimizations, RAMMER can provide significant additional performance improvement, especially for DNN inference workloads.

RAMMER is hardware neutral. The abstractions proposed, such as *r*Task, *r*Operator and vEU are applicable to any massively parallel computational devices with homogeneous execution units. This includes almost all the computational devices proposed for DNN workloads. In this paper, in addition to describe in detail how RAMMER is implemented on NVIDIA GPUs, we will also discuss our experience retargeting RAMMER for several alternative computing devices.

We have implemented RAMMER with 52k lines of C++ code and open-sourced the code[1]. Our evaluation on 6 DNN models shows that RAMMER significantly outperforms state-of-the-art compilers like XLA and TVM on both NVIDIA and AMD GPUs, with up to 20.1× speedup. RAMMER even outperforms TensorRT [13], a vendor optimized DNN inference library from NVIDIA, with up to 3.1× gain.

Our experience on RAMMER strongly suggests that the current industry-prevalent practice of vendor supplying highly optimized DNN operator implementations in a library form (such as cuDNN and MKL-DNN) is sub-optimal. This practice will incur significant efficiency cost for DNN workloads. The situation will become even worse in the coming years as modern accelerators keep on increasing the available hardware parallelism while new DNN architectures strive to save computation by replacing larger operators with many smaller ones [49,54]. We recommend vendors to supply optimized implementations in other forms, such as our proposed *r*Operator and vEU abstractions, in order to enable holistic optimization that can fully utilize hardware resources.

2 Motivation

In this section, we highlight some results to illustrate the limitation of the two-layer design of existing deep learning frameworks. Without loss of generality, we experiment with TensorFlow [18], a state-of-the-art DNN framework, on an NVIDIA GPU, using the same settings as in §5.

[1] https://github.com/microsoft/nnfusion

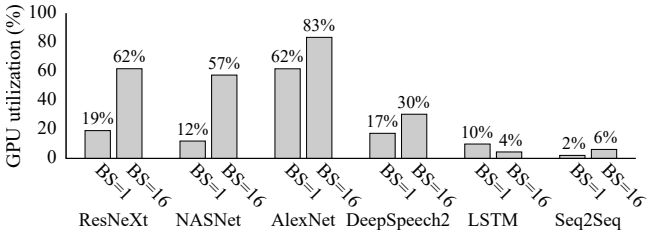


Figure 1: The average GPU utilization on different DNN model with different batch size (BS). The utilization only accounts for kernel execution, excluding other stages like operator emitting.

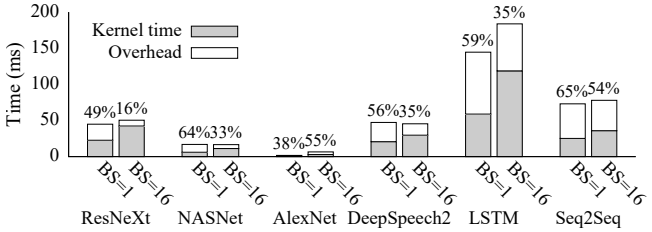


Figure 2: The average kernel time and end-to-end execution time on different DNN model with different batch size (BS).

Hardware-managed intra-operator scheduling leads to low GPU utilization. The two-layer design delegates the intra-operator scheduling to the hardware scheduler in an accelerator like GPU. Figure 1 shows that such an approach could lead to low GPU utilization across different DNN models. When the batch size is 1, the GPU utilization could be as low as 2% for Seq2Seq model. Even when the batch size is increased to 16, the average GPU utilization across the 6 models is merely 40% [2]. To improve the scheduling efficiency, modern GPUs support the multi-streaming mechanism that allows independent operators to run concurrently. However, our measurement in §5 shows that multi-streaming often hurts rather than improves the overall performance.

High inter-operator scheduling overheads. The two-layer approach also incurs a higher inter-operator scheduling overheads. Here, we regard the time not spent doing actual computation in the GPU as the overhead for inter-operator scheduling. This overhead includes various operations to support operator emitting, including kernel launching, context initialization, communication between host and GPU, and so on. The percentage shown above each bar in Figure 2 depicts how much time the DNN model is *not* spent in the actual GPU computation. From the figure it is clear that the overhead of inter-operator scheduling is quite significant. When batch size is 1, the average overhead is 55% across the 6 DNN models. Increasing the batch size to 16 slightly improves the situation, while the overhead is still not negligible (between 16% and 55%). Modern DNN compilers, including the one in TensorFlow, employs a technique called kernel fusion [17,23], which merges several DNN operators into a single one when allowed. However, our results in §5 show that this technique cannot reduce the overhead significantly.

[2]Note that the LSTM's GPU utilization is slightly higher when batch size is 1 compared to that when batch size is 16, because TensorFlow uses different kernel implementations of GEMM for different batch sizes.

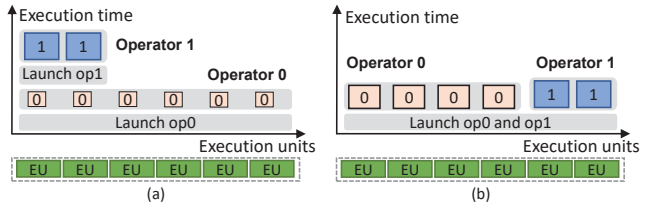


Figure 3: An illustration of (a) the inefficiency scheduling in existing approach; and (b) an optimized scheduling plan.

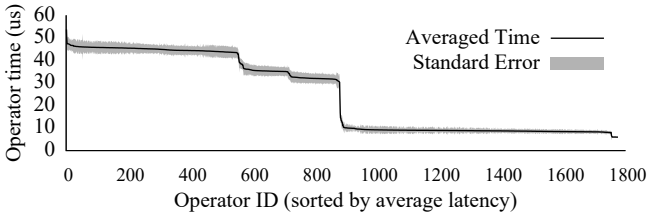


Figure 4: The profiled kernel time of all the operators in ResNeXt model. Each data point ran 1,000 times

Interplay between inter- and intra-operator scheduling. Separating scheduling into two layers ignores the subtle interplay between inter-operator and intra-operator scheduling, which may lead to suboptimal performance. For example, Figure 3(a) shows two independent operators being scheduled to a GPU. For operator 0, to maximize its performance, the system may choose the fastest implementation with a high degree of parallelism. Thus operator 0 could greedily span all the parallel execution units (EUs) of an accelerator (in this case the streaming multiprocessors of the GPU), while each EU may not be fully utilized. Since operator 0 occupies all the EUs, operator 1 has to wait for available resource. A better scheduler could reduce the degree of parallelism of operator 0 to increase the level of inter-operator parallelism, by mapping operator 1 alongside operator 0, as illustrated by Figure 3(b). We will discuss more details of this issue in §3.3 and §5.

Opportunities. Given the fundamental limitations of the two-layer design observed above, it is desirable to manage the scheduling of inter and intra-operator together. However, a naive implementation of this approach may incur even higher overheads than the already significant inter-operator scheduling overheads. Fortunately, most DNN's DFG is available at

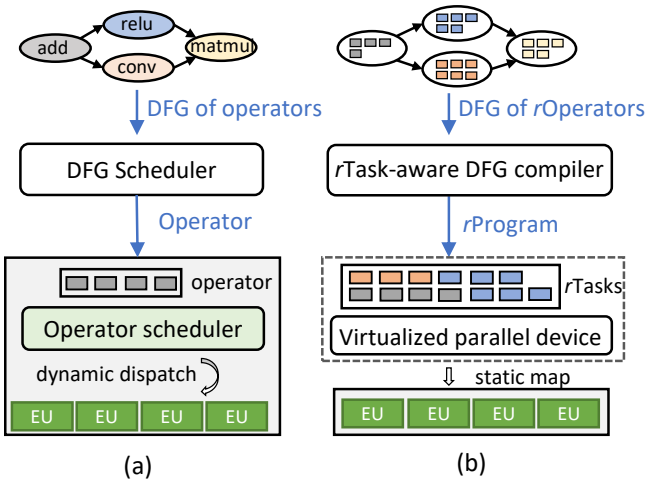


Figure 5: System overview of DNN computation in (a) existing DNN frameworks, and (b) RAMMER, where each node in a DFG is an *r*Operator that explicitly exposes the intra-operator parallelism through *r*Task; Rather than dynamically scheduling each *r*Operator, RAMMER compiles the DFG into a static execution plan called *r*Program (composed of *r*Tasks) and maps it to hardware by a software device abstraction called vDevice.

the compile time, and the operators often exhibit deterministic performance, therefore, their execution times can be obtained through compile time profiling [45]. For example, Figure 4 shows the averaged GPU kernel time and the variance of all the operators in the ResNeXt [49] model. The kernel run-time weighted average of standard deviations among all operators is only 7%. This allows us to move the scheduling from run-time to compile-time, by generating an offline schedule plan to reduce runtime overhead.

3 RAMMER's Design

The observations in §2 motivate RAMMER, a DNN compiler framework that manages both inter and intra-operator scheduling. Figure 5 shows the key differences between an existing deep learning framework and RAMMER. First, the input to RAMMER is a data-flow graph where a node is an *rOperator*, rather than a traditional operator. An *r*Operator explicitly exposes *r*Task, a fine-grained computation unit that could run on a parallel execution unit in an accelerator. We discuss details of *r*Task in §3.1. Second, instead of separating the two-layer scheduling between software and hardware, RAMMER introduces *rTask-aware DFG compiler* to manage the inter and intra-operator scheduling in one place. The *r*Task-aware DFG compiler will generate a static execution plan for runtime execution. Often, it is not efficient or not possible to pack the entire DNN computation in a single accelerator device invocation. Therefore, the execution plan is breaking into

```
1 interface Operator { void compute(); };
2 interface rOperator {
3   void compute_rtask(size_t rtask_id);
4   size_t get_total_rtask_num();
5 };
```

Figure 6: The execution interfaces of traditional operator and *r*Operator. More details in §4.

multiple *r*Programs, each contains a piece of computation to be carried out on the hardware. Instead of emitting one operator at a time for an accelerator, RAMMER emits an *rProgram* at a time. The details of the *r*Task-aware DFG compiler will be discussed in §3.3. To carry out the execution plan, RAMMER abstracts a hardware accelerator as a *virtualized parallel device* (vDevice), which includes multiple virtualized execution units (vEUs). The vDevice provides the scheduling and synchronization capabilities at the *r*Task level so that the *r*Program can be mapped to the corresponding vEUs at compile time. The vEUs, together with the vDevice will be mapped to the hardware at runtime. We introduce virtualized device in §3.2.

3.1 *r*Operator

An *r*Operator is defined as a group of independent, homogeneous ***r*Task**s (short for RAMMER task), where an *r*Task is the minimum computation unit in an operator to be executed on a processing element of the accelerator device. The concept of *r*Task naturally aligns with the parallel architecture of DNN accelerators, e.g., the SIMD architecture of GPU. To maximize efficiency, the computation on such an accelerator needs to be divided into multiple parallel (homogeneous) tasks. Each of these parallel tasks can be represented by an *r*Task, thereby exposing the intra-operator parallelism not only to the underlying hardware, but to the RAMMER compiler. Given that an *r*Task is logically identical to a parallel task, RAMMER relies on external tools to partition an *r*Operator into *r*Tasks (e.g., TVM [23]). In another word, RAMMER uses external heuristics to decide a reasonable granularity of *r*Task.

As a concrete example, a matrix multiplication operator can be divided into multiple homogeneous *r*Tasks, each computes a tile of the output matrix, while the tiling strategy is assumed to be given. If a complicated DNN operator can hardly be divided into independent homogeneous *r*Tasks (e.g., SeparableConv2D [7]), it can be represented as multiple dependent *r*Operators, each can be partitioned into *r*Tasks.

An *r*Task is indexed by a logical `rtask_id`. The *r*Tasks in an *r*Operator are numbered continuously. To execute an *r*Task, the parallel execution unit could call the `compute_rtask()` interface (line 3 Figure 6). To generate an *r*Program, RAMMER needs to know the total number of *r*Tasks in an operator. This is available through the interface `get_total_rtask_num()`. In contrast, a traditional opera-

tor has only one interface `compute()` (line 1 Figure 6). The implementation of an *r*Operator is called *r*Kernel, which realizes the concrete *r*Task computation logics and decides the total number of *r*Tasks. One *r*Operator might have multiple versions of *r*Kernels based on different tiling strategies, e.g., trading off between resource efficiency and overall execution time.

The *r*Operator abstraction allows RAMMER to expose both inter- and intra-operator parallelisms. This opens up a new space to optimize DNN computation holistically.

3.2 Virtualized Parallel Device

Modern accelerators do not provide interfaces to map an *r*Task to a desired execution unit directly. For example, a GPU only allows to execute one operator (in the form of a kernel) at a time. To address this challenge, RAMMER abstracts a hardware accelerator as a software-managed virtual device called *virtualized parallel device* (*vDevice*). A vDevice further presents multiple parallel *virtual execution units* (*vEU*s), each of them can execute *r*Tasks independently.

With vDevice, RAMMER organizes the computation of *r*Task-aware DFG as an *r*Program on a vDevice. An *r*Program is represented as two dimensional array of *r*Task `prog[vEU_id][order]`, where `vEU_id` denotes the vEU the *r*Task is assigned to, and `order` denotes the execution order of the *r*Task in this vEU. For example, `prog[0][0]` denotes the first *r*Task to be executed in vEU `0`. To ensure the correct execution of dependent *r*Tasks in a plan, RAMMER introduces ***barrier-rTask***. A barrier-*r*Task takes the argument of a list of pairs `<vEU_id, order>`. The barrier-*r*Task will wait until the completion of all *r*Tasks indexed by each pair. The barrier-*r*Task provides a fine-grained synchronization mechanism to enable *r*Task schedule plan execution.

For the execution of DNN computation, a vDevice needs to be mapped to a physical accelerator at runtime. We will discuss how RAMMER implements the mapping of vDevice to different hardware accelerators in §4.

3.3 *r*Task-aware DFG Compiler

The *r*Task abstraction and the fine-grained *r*Task execution capability exposed by the vDevice open up a large optimization space. RAMMER aims to generate a high-quality schedule in this space, represented as a sequence of *r*Programs. To this end, the *r*Task-aware DFG compiler separates the scheduling mechanism from its policy. On the mechanism side, it provides two capabilities: (1) Two scheduling interfaces for a policy to generate an execution plan. (2) A profiler to supply profiling information requested by a scheduling policy.

Scheduling interfaces. RAMMER's *r*Task-aware DFG compiler introduces two scheduling interfaces, `Append` and `Wait`. `Append(task_uid, vEU_id)` assigns an *r*Task from an operator to the specified vEU in a sequential order. Here `task_uid` is a global identifier for an *r*Task, which is essentially the operator `id` combined with the `rtask_id` within the operator. The second API, namely `Wait(wtask_uid, list<task_uid>)`, allows an *r*Task specified by `wtask_uid` to wait for *r*Tasks in `list<task_uid>`. The `Wait` interface will implicitly `Append` a *barrier-r*Task (discussed in §3.2) right before the *r*Task `wtask_uid`. As an optimization, when waiting for multiple consecutive *r*Tasks $r_1, r_2, ..., r_n$ sequentially appended to the same vEU, the *r*Task only need to include the last one, i.e., r_n, in the waiting list.

Algorithm 1: Wavefront Scheduling Policy

```
   Data: G: DFG of rOperator, D: vDevice
   Result: Plans: rPrograms
1  Function Schedule(G, D):
2      P_curr = {};
3      for W = Wavefront(G) do
4          P_1 = ScheduleWave(W, P_curr, D);
5          P_2 = ScheduleWave(W, {}, D);
6          if time(P_1) ≤ time(P_curr) + time(P_2) then
7              P_curr = P_1;
8          else
9              Plans.push_back(P_curr);
10             P_curr = P_2;
11     return Plans;
12 Function ScheduleWave(W, P, D):
13     SelectRKernels(W, P);
14     for op ∈ W do
15         for r ∈ op.rTasks do
16             vEU = SelectvEU(op, P, D);
17             P.Wait(r, Predecessor(op).rTasks);
18             P.Append(r, vEU);
19     return P;
```

Compile-time profiling. RAMMER profiler provides the following three types of information: 1) individual *r*Task execution time on a vEU; 2) resource usage of an *r*Task such as the local memory or registers used and 3) the overall execution time of an *r*Program. This profiling information can guide a policy to generate an efficient scheduling plan.

Scheduling policy. Algorithm 1 illustrates how to use the above scheduling interfaces and the profiler to implement a scheduling policy to exploit both inter- and intra-operator parallelisms. This policy takes an *r*Task-aware DFG and schedules operators in *waves* [37]. The operators in a wave are the fringe nodes of a breadth-first-search on the DFG. The policy will include a wave's operators in the current *r*Program if the profiling results (denoted by `time()`) suggest it will reduce the total execution time. Otherwise, the policy will create a separate *r*Program (line 2-10).

First of all, we assume that each *r*Operator has one or more implementations called *r*Kernels, each *r*Kernel is a way to break the operator into *r*Tasks with different resource and runtime trade-offs. Among the *r*Kernels of a particular *r*Operator,

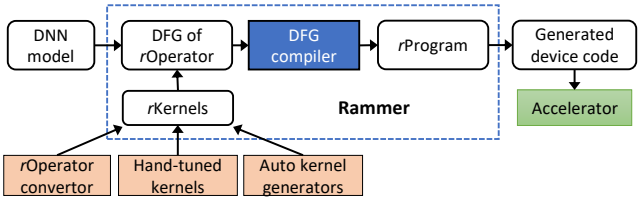


Figure 7: Overall workflow of RAMMER.

there is the *fastest* one with the smallest runtime, and there is the most *efficient* one with the smallest product of runtime and the total number of *r*Tasks.

For each wave, the policy selects the implementations of the operators through `SelectRKernels()` (line 13), with the following heuristics: If combine all the *r*Tasks in the wave with the *fastest* operator implementations still cannot occupy all the parallel execution units in the accelerator, the policy will just select them. Otherwise, the policy will find the most *efficient* *r*Kernels and then perform a profiling. The policy will choose these *r*Kernels if the profiling results show better execution time, otherwise it will stick with the fastest *r*Kernels. This heuristic considers the interplay between the inter- and intra-operator scheduling by evaluating the *r*Operators (and their *r*Tasks) in a wave, instead of individually. After the *r*Kernel selection, the policy calls `SelectvEU()` to decide which vEU an *r*Task should be scheduled to (line 16). Given the current *r*Program `P`, `SelectvEU()` chooses the vEU that can execute the *r*Task at the earliest, based on the profiled execution time of each *r*Task in `P`. Finally, the policy calls `Wait()` to ensure *r*Task level dependency (derived from the DFG) and `Append()` to assign the *r*Task to the selected vEU (line 17-18). The policy in Algorithm 1 demonstrates how RAMMER separates the scheduling mechanism from scheduling policy. As shown in §5, this simple policy can already outperform the state-of-the-art, sometimes significantly. We envision the proposed scheduling mechanism could enable future research on more advanced scheduling policies to further explore the optimization space.

4 Implementation

We implement RAMMER with 52k lines of C++ code, including 3k lines of code for the core compiler and scheduling function. The input of RAMMER is a DNN model in either TensorFlow [18] frozen graph, TorchScript [16] or ONNX [14] format. RAMMER first converts the input model into a DFG of *r*Operators. Since the input model is often not optimized, like other compilers, we also implemented common graph optimizations such as constant folding, common sub-expression elimination, pattern-based kernel fusion, etc. For each *r*Operator from an optimized DFG, RAMMER loads one or multiple versions of *r*Kernel implementations from different sources, e.g., auto-kernel generators [23], hand-tuned kernels, or converted from existing operators in other frameworks. RAMMER compiler will then partition the DFG into sub-graphs (e.g., based on the policy in Algorithm 1) and compile each of them as an *r*Program. As an output, each *r*Program is further generated as a device code (e.g., GPU kernels) that runs on the accelerator. Figure 7 summarizes the overall workflow of RAMMER.

```
1  __device__ void matmul_rTask(float *A, float *B,
2      float *C, size_t rtask_id) {
3    size_t tile_x = rtask_id / (M/32);
4    size_t tile_y = rtask_id % (N/32);
5    size_t i = threadIdx.x/32 + tile_x*32;
6    size_t j = threadIdx.x%32 + tile_y*32;
7    C[i][j] = 0;
8    for (size_t k = 0; k < K; k++)
9      C[i][j] += A[i][k] * B[k][j];
10 }
11
12 class MatmulROperator {
13  __device__ void compute_rtask(size_t rtask_id){
14    matmul_rTask(input0,input1,output0,rtask_id);}
15  size_t get_total_rtask_num(){return M/32*N/32);}
16 };
```

Figure 8: A CUDA implementation of a naive matrix multiplication with the *r*Operator abstraction.

In the rest of this section, we describe the details about RAMMER's implementation for CUDA GPU. We focus on NVIDIA GPUs and the CUDA eco-system because they are the most widely used accelerators for DNN. To demonstrate that the vDevice abstraction enables RAMMER compiler to support different accelerators with an uniform interface, we will also briefly describe our experience with other DNN accelerators, including AMD GPUs and Graphcore IPU, at the end of this section.

4.1 RAMMER on NVIDIA CUDA GPUs

An NVIDIA GPU usually consists of tens to hundreds of *streaming multiprocessors (SM)*, each containing tens of cores. Computation on SM follows the Single Instruction Multiple Thread (SIMT) model. In this paper we assume the readers are familiar with the basic concepts of CUDA [4], the programming paradigm introduced by NVIDIA to program their GPUs. A single CUDA program (often referred to as a CUDA kernel) groups multiple threads into *block*s, each thread-block is assigned to run on an SM, where the scheduling is performed by GPU hardware. RAMMER naturally maps each vEU to an SM and implements an *r*Task as a thread-block.

4.1.1 *r*Operator in CUDA

Figure 8 shows a naive CUDA implementation of an *r*Operator that multiplies a $M \times K$ matrix A by a $K \times N$ matrix B. For simplicity, we assume M and N are evenly divisible

by 32. In the code, each *r*Task computes a 32×32 tile of the output matrix *C*. Line 1-10 in Figure 8 shows the computation of one thread in one *r*Task. The thread uses `rtask_id`, a RAMMER assigned id to identify the tile to be computed by this *r*Task (line 3-4), and uses `threadIdx`, a CUDA built-in thread index to identify the data element to be computed (line 5-6) by this thread. The identified element is then computed in line 7-9. Line 13 shows the interface exposed by this *r*Operator, which will be called by a vEU's parallel thread. The total *r*Tasks needed in this operator is determined by the matrix dimension *M* and *N*, and can be obtained through the `get_total_rtask_num` interface (line 16). The key difference between code in Figure 8 and a traditional CUDA code is that an *r*Task uses `rtask_id`, a logical index controlled by RAMMER, instead of `blockIdx`, a built-in thread-block index controlled by the GPU's hardware scheduler. This enables RAMMER to map an *r*Task to a desired vEU by executing `compute_rtask()` with a proper `rtask_id`. Note that the code shown in Figure 8 is for illustrative purpose. The evaluation shown in §5 uses a more complicated tiled version of matrix multiplication *r*Operator, which further improves the performance through carefully exploiting GPU memory hierarchy, e.g., shared memory and registers [36, 41].

4.1.2 vDevice and vEU on CUDA GPU

On a CUDA GPU, the intra-operator scheduling is usually managed by the GPU's built-in scheduler. To bypass the built-in scheduler, RAMMER leverages a persistent thread-block (PTB) [29] to implement a vEU in a vDevice. PTB is a thread-block containing a group of continuously running threads, where RAMMER is able to "pin" the PTB to the desired SM. Given an *r*Program, each thread in the PTB (and hence the vEU) executes the `compute_rtask()` according to the sequence specified by the *r*Program. To execute the `compute_rtask()` from multiple *r*Tasks continuously in a PTB, a function qualifier `__device__` is required by CUDA for `comptue_rtask()` and any sub functions executed therein (e.g., line 1 and 13 in Figure 8).

Figure 9 illustrates the CUDA code for a vDevice with two vEUs, i.e., a CUDA kernel function with two PTBs. This vDevice executes an *r*Program compiled from a DFG with three *r*Operators: a `Matmul`, a `Relu`, and a `Conv`. Specified by the execution plan, the vDevice executes two *r*Tasks of the `Matmul` operator on vEU 0, and in parallel it also runs four *r*Tasks of the `Relu` operator on vEU 1. Then a global barrier is inserted to the two vEUs, each runs a barrier-*r*Task: vEU 0 waits for the 4*th* *r*Task on vEU 1, and vEU 1 waits for the 2*nd* *r*Task on vEU 0. Finally, the vDevice executes two *r*Tasks of the `Conv` operator on the two vEUs respectively. On each vEU, RAMMER runs the *r*Tasks sequentially in a code branch, executed only if the current vEU Id matches the one generated by the *r*Program.

Before a lengthy DNN computation, RAMMER dispatches

```
1  // config: <<<(vEU_size,1,1), (vEU#,1,1)>>>
2  __global__ void vdevice_run() {
3    if (Get_vEU_Id() == 0) { // vEU 0
4      MatmulrTaskOp.compute_rtask(0);
5      MatmulrTaskOp.compute_rtask(1);
6      // wait the rTask on vEU 1 with order=3
7      BarrierTask({<1, 3>}).compute_rtask();
8      Conv2DrTaskOp.compute_rtask(0);
9    }
10   else if (Get_vEU_Id() == 1) { // vEU 1
11     for (auto i : 4)
12       RelurTaskOp.compute_task(i);
13     // wait the rTask on vEU 0 with order=1
14     BarrierTask({<0, 1>}).compute_rtask();
15     Conv2DrTaskOp.compute_rtask(1);
16   }
17 }
```

Figure 9: The CUDA code for a vDevice with two vEUs.

each vEU (implemented by a PTB) to a desired SM through the GPU scheduler [48]. To improve hardware utilization, an SM can run multiple vEUs (PTBs) concurrently. Since CUDA uses a SIMT model, all vEU are homogeneous, the number of vEUs an SM can support depends on the most demanding *r*Task across all the vEUs, i.e., the *r*Task that requires the most thread number, register number, shared memory size, etc. In practice, we set the number of vEUs on each SM according to the maximum active PTB number provided by the CUDA compiler `nvcc` [5]. With the vDevice abstraction, the optimizations in RAMMER become hardware agnostic.

4.1.3 Executing *r*Task on vEU in CUDA

Executing heterogeneous *r*Tasks. In a CUDA kernel, the number of threads in a thread block is fixed in the entire execution lifecycle. This force RAMMER to require that all the *r*Tasks on a vEU to run on with the same number of persistent threads. In practice, different *r*Operators may use different number of threads to balance parallelism and per-thread resource usage. To address this problem, RAMMER sets the number of threads of a vEU to be the maximum number of threads used by an *r*Task in the vEU. For an *r*Task with less threads, RAMMER inserts early-exit logic in the extra threads to skip the unnecessary (and invalid) execution. However, early-exit may lead to dead-lock: a global barrier might never return because early-exit logic may skip the barrier. To avoid this issue, RAMMER can leverage the CUDA cooperative group primitives [3], which explicitly controls the scope of threads during a synchronization.

Implementing barrier-*r*Task. To implement an efficient barrier-*r*Task, RAMMER introduces a step array, where each element is an integer tracking the number of finished *r*Tasks in each vEU. When finished, an *r*Task will use its first thread to increase the corresponding element in the step array by 1. When waiting for a list of *r*Tasks on *N* vEUs, a barrier-

*r*Task uses its first N threads to poll on the corresponding elements in the step array until the steps are larger than the `orders` of those *r*Tasks. After that, the barrier-*r*Task calls `__syncthreads` to ensure all threads in this vEU are ready to run the next *r*Task.

4.1.4 Transforming Legacy CUDA Operators

Many operators for DNN are already available as CUDA kernel code. To reduce development efforts, RAMMER introduces a source-to-source converter to transform a legacy CUDA operator into an *r*Operator. The key insight of the converter lies on the facts that to exploit the intra-operator parallelism, legacy CUDA operators are also implemented as thread-blocks, although they use `blockIdx` and let CUDA GPU hardware control the intra-operator scheduling directly. *r*Operator can just compute the desired `blockIdx` from `rtask_id` without changing computation logic in the legacy kernel.

One challenge in this transformation is that the thread-blocks in existing operator could be laid out in 1, 2, or 3-dimensional shape, while in a vEU threads are laid out in a 1-dimensional shape. This means our vEU needs to support *r*Task with different threads shapes. For example, Figure 10 illustrates a vEU executing two *r*Tasks with the thread shapes of $[2 \times 2]$ and $[2 \times 3]$ respectively. Our solution is to stick to a 1-D persistent thread shape for a vEU, and apply a thread index remapping to compute the desired `threadIdx` in the legacy kernel with the vEU's 1-D `threadIdx`. Notice that, as discussed before, the number of threads of a vEU is the maximum number of threads of all *r*Task in the vEU, so that such a remapping is always possible. For example, in Figure 10 we configure the vEU with $[1 \times 6]$ persistent threads. When executing *r*Task 0 with a legacy $[2 \times 2]$ thread shape, RAMMER remaps the $[2 \times 2]$ shape to the vEU's $[1 \times 6]$ thread.

In summary, to convert a legacy DNN operator to an *r*Operator, one needs to remap thread and block index, implement the early-exit logic, and use CUDA cooperative group primitive to support local barrier on the active (i.e. not early-exited) threads. RAMMER implements these changes by inserting a compiler-generated code segment at the entry point of the legacy operator kernel code. With these modifications, RAMMER can preserve the legacy operator implementation, and reuse it as an *r*Task operator. In RAMMER, we have transformed and implemented total 150 *r*Kernels for 70 *r*Operators.

4.2 RAMMER on Other Accelerators

The design of RAMMER is not limited to CUDA and NVIDIA GPUs. In fact, our *r*Task, *r*Operator and vEU abstractions are applicable to any massively parallel computational devices with homogeneous execution units, including most of the devices that used for DNN computation. In this section, we discuss how to port RAMMER to support other devices.

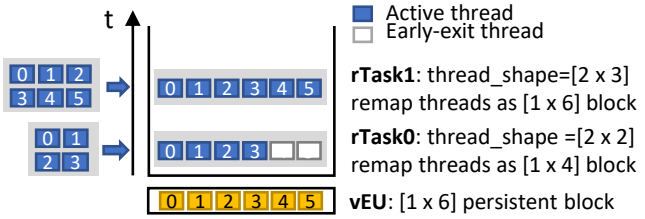


Figure 10: Executing two heterogeneous *r*Tasks on a vEU.

4.2.1 RAMMER on AMD GPUs

AMD GPUs are similar to NVIDIA GPUs, which also consist of many parallel execution units called *compute units (CU)*. AMD GPU has a HIP programming model [8], which is similar to CUDA. AMD provides a `hipify` tool that can convert a CUDA kernel to a HIP kernel. `hipify` can help convert most CUDA *r*Operators to the HIP version. Some CUDA kernel configurations, such as the number of threads per thread-block and size of local memory, are not optimized for AMD GPUs due to the minor architecture differences. We re-implemented 41 *r*Kernels for AMD GPUs for better performance. `hipify` can also convert the CUDA implementation of vDevice (i.e., PTBs) to the HIP version. The only exception is that AMD GPUs do not support cooperative group primitives. To address this issue, we introduce a new API in *r*Operator to provide the number of (block-wise) synchronizations S (i.e. calls to `__syncthreads`). For early-exit threads, instead of exit immediately, RAMMER will insert code to call the `__syncthreads` primitive S times.

4.2.2 RAMMER on Graphcore IPU

The Graphcore IPU (Intelligence Processing Unit) [10] is a state-of-the-art DNN accelerator with an architecture quite different from GPUs. IPU is a massively parallel MIMD processor with a bulk-synchronous-parallel (BSP) communication model. Each IPU contains 1,216 parallel processing units called *tiles*; a tile consists of a hyper-threaded computing core plus 256 KB of local memory. DNN computation on an IPU is explicitly programmed as a data-flow graph, where each vertex implements the code executed on a tile and each edge depicts the data transfers between vertices. The IPU compiler is responsible for mapping each vertex to a tile.

RAMMER's *r*Task abstraction can also map to IPU's MIMD model: a vEU can map to a tile and a vertex can be treated as an *r*Task. Thus, an *r*Operator on IPU can be implemented as a set of vertices. More importantly, IPU compiler allows to control the vertex-tile mapping at compile-time. This provides the core functionality required in vDevice abstraction. Restricted by the hardware BSP model, IPU does not provide a fine-grained synchronization mechanism. We therefore implement barrier-*r*Task with a global barrier, which may reduce scheduling space for RAMMER. Even with this limitation, RAMMER

Model	Dataset	Model Type	Note
ResNeXt	CIFAR-10	Computer Vision	layers: 29, cardinality: 16, bottleneck width: 64d (16×64d, paper parameter)
NASNet	CIFAR-10	Computer Vision	repeated cells: 6, filters: 768 (6@768, paper parameter)
AlexNet	ImageNet	Computer Vision	(paper parameter)
DeepSpeech2	LibriSpeech	Speech	input length: 300; CNN layer: 2; RNN: type: uni-LSTM, layer: 7, hidden size: 256
LSTM (-TC)	synthetic	Language Model	input length: 100, hidden size: 256, layer: 10
Seq2Seq (-NMT)	synthetic	Language Model	Encoder: input length: 100, type: uni-LSTM, hidden size: 128, layer: 8 Decoder: output length: 30, type: uni-LSTM, hidden size: 128, layer: 4

Table 1: Deep learning models and datasets.

still can schedule *r*Tasks of different operators at the same computing step to increase utilization. To evaluate RAMMER, we implemented total 15 *r*Operators and 18 *r*Kernels.

4.2.3 RAMMER on x86 CPUs

We also implemented RAMMER on multi-core x86 CPUs. However, we see little performance benefit of adopting the RAMMER abstractions on x86-based platforms. On x86, the operator runtime is high due to the relatively low performance of x86 cores for numerical computations, and the small number of cores can be fully occupied by almost any DNN operators. Moreover, scheduling overhead is not significant because kernel launch is just a regular function call. Therefore, RAMMER cannot provide additional benefit compared with the traditional two-layered scheduling approach.

5 Evaluation

In this section, we present the detailed evaluation results to demonstrate the effectiveness of RAMMER with comparison with other state-of-the-art frameworks.

5.1 Experimental Setup

Machine environment. We evaluated RAMMER on three servers with different accelerators equipped. The CUDA GPU evaluations use an Azure NC24s_v3 VM equipped with Intel Xeon E5-2690v4 CPUs and 4 NVIDIA Tesla V100 (16GB) GPUs, with Ubuntu 16.04, CUDA 10.0 and cuDNN 7.6.5. The AMD ROCm GPU evaluations use a server equipped with Intel Xeon CPU E5-2640 v4 CPU and 2 AMD Radeon Instinct MI50 (16GB) GPUs, installed with Ubuntu 18.04 and ROCm 3.1.1 [1]. The IPU evaluations use an Azure ND40s_v3 preview VM equipped with Intel Xeon Platinum 8168 CPUs and 16 IPUs with Poplar-sdk 1.0.

We compare RAMMER with other DNN frameworks and compilers, including TensorFlow (v1.15.2) representing the state-of-the-art DNN framework, TVM (v0.7) [23] and TensorFlow-XLA representing the state-of-the-art DNN compilers, and TensorRT (v7.0) (with TensorFlow integration version), a vendor-specific inference library for NVIDIA GPUs.

Benchmarks and datasets. Our evaluation is performed using a set of representative DNN models that covers typical deep neural architectures such as CNN and RNN; and different application domains including image, NLP and speech. Among them, ResNeXt [49] is an improved version of ResNet [30]; NASNet [54] is a state-of-the-art CNN model obtained by the neural architecture search; AlexNet [35] represents a classic CNN model with a simple architecture. LSTM-TC [31] is an RNN model for text classification; DeepSpeech2 [19] is a representative speech recognition model; and Seq2Seq [46] is for neural machine translation. All the implementations of these benchmarks, including the *r*Kernels used in each model, are available in our artifact evaluation repository[3].

We focus our evaluation on model inference. There is no fundamental reason limiting RAMMER from model training, except that supporting training requires us to develop more operators. We evaluate these models on a set of datasets including CIFAR-10 [2], ImageNet [26], LibriSpeech [11] and synthetic datasets. Table 1 lists the models, hyper-parameters, and the corresponding datasets used. All performance numbers in our experiments are averages over 1,000 runs; in all cases we observed very little variations.

5.2 Evaluation on CUDA GPUs

This section answers the following questions: 1) How does RAMMER perform comparing with the state-of-the-art DNN frameworks or compilers? 2) How well does RAMMER utilize the GPU's parallel resource? 3) How much does RAMMER reduce the runtime scheduling overhead? 4) How much performance gain comes from RAMMER's scheduling leveraging both the intra and inter operator parallelism? 5) How effective is the fine-grained synchronization in improving the overall performance?

5.2.1 End-to-end Performance

We first demonstrate the end-to-end efficiency of RAMMER by comparing with TensorFlow (TF), TensorFlow-XLA (TF-

[3] https://github.com/microsoft/nnfusion/tree/osdi20_artifact/artifacts

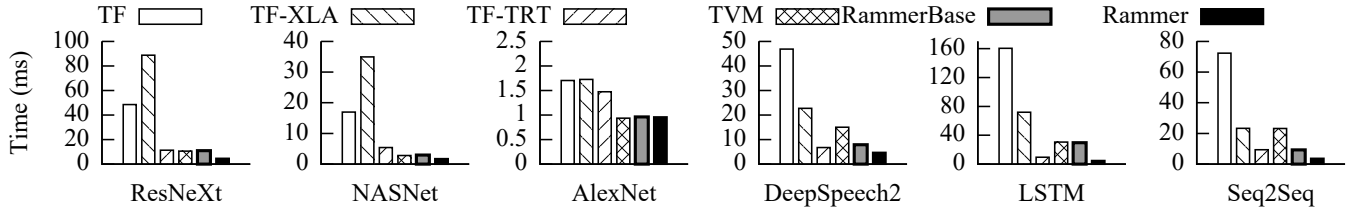


Figure 11: End-to-end model inference time with batch size of 1 on NVIDIA V100 GPU.

XLA), TVM and TensorRT (TF-TRT). To show the benefit of the abstractions introduced in RAMMER, we create a baseline version of RAMMER (called RAMMERBASE), which only implements the optimizations similar to those in existing compilers and still uses a two-layered scheduling approach. Thus, RAMMERBASE can be treated as just another regular DNN compiler implemented in the same codebase of RAMMER. Figure 11 shows the execution time of the benchmarks with batch size of 1.

First, RAMMER significantly outperforms TF by 14.29× on average, and up to 33.94× for the LSTM-TC model. The performance improvement of RAMMER against TF is mainly because TF suffers from heavy runtime scheduling overhead at DFG level, especially when the individual operator's execution time is relatively short, as is the case in small batch inference. TF-XLA, as a DNN compiler, can improve TF's performance through DFG level optimizations (e.g., operator fusion) and operator-level code specializations (e.g. customized kernel generation). However, it still cannot fully avoid scheduling overhead, which leads to an average of 11.25× (up to 20.12×) performance gap compared to RAMMER. We observed that TF-XLA incurs even higher overhead for some CNN models such as ResNeXt and NASNet compared with TF. TVM, as another state-of-the-art DNN compiler, mainly leverages a kernel tuning technique to generate a specialized kernel for each operator. In our evaluation, TVM tunes 1,000 steps and chooses the fastest kernel for each operator. With such specialized optimization, TVM can improve the performance significantly compared with TF and TF-XLA. Still, RAMMER can outperform TVM by 3.48× on average and up to 6.46×. Even though TVM can make individual operator run faster through tuning, it still lacks the capability to leverage the fine-grained parallelism as RAMMER. An exception is that, for AlexNet, RAMMER can only achieve comparable performance with TVM. This is mainly because AlexNet, being one of the earliest modern DNN models, can be easily optimized due to its simple sequential model architecture and relatively fewer, but larger operators. Finally, TensorRT is a specialized DNN inference library with highly optimized operators provided by NVIDIA. We use its official TensorFlow-integration version (TF-TRT) to compile and run our models, as its stand-alone version fails to directly compile these benchmarks. However, for RNN models like DeepSpeech2, LSTM-TC and Seq2Seq-NMT, TF-TRT failed

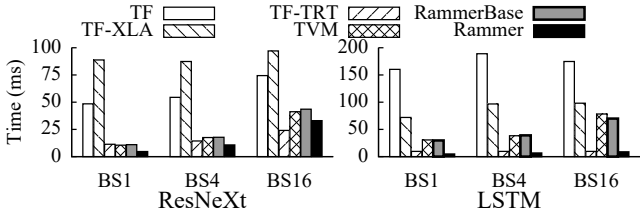


Figure 12: End-to-end model inference time with different batch sizes (BS).

to produce results after compiling for over 50 hours. Thus, we reimplemented these three models with the TensorRT native APIs. Our evaluation shows that RAMMER can outperform the vendor optimized TensorRT on all the benchmarks, with an averaged 2.18× and up to 3.09× lower latency. Finally, compared to RAMMERBASE, RAMMER can further improve the end-to-end performance by 2.59× and up to 6.29×.

Performance with different batch sizes. We also evaluate RAMMER's performance with larger batch sizes. Figure 12 shows the performance comparison on two representative CNN and RNN models, i.e., ResNeXt and LSTM-TC, with batch sizes of 4 and 16. We limit our benchmarks in this test due to the cost of developing optimized *r*Operator kernels for RAMMER: we have to hunt for efficient open-sourced operator kernel implementations or perform tuning by hand or through automatic tuning tools, which is time consuming. As it shows, using larger batch sizes can reduce scheduling overhead in existing frameworks due to the increased per-operator execution time. Even so, RAMMER can still outperform all the systems except for TensorRT on the ResNeXt model with batch size of 16. For this case, TensorRT uses some operators whose source codes are not publicly available, and our implementations do not yet match their performance. In fact, implementing operators to match the performance of close-sourced kernels is one of the major challenges for RAMMER. Compared to the other open source frameworks and compilers, RAMMER has a significant gain. For example, when using batch size of 16, RAMMER can outperform TF by 2.25×, and TVM by 1.25× on ResNeXt. For the LSTM-TC model, RAMMER can get 20.08× and 9.0× performance gains compared with TF and TVM respectively.

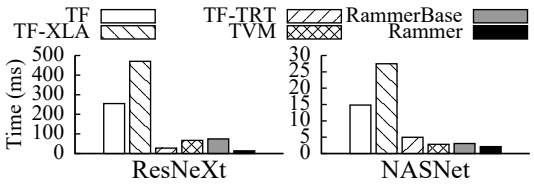


Figure 13: End-to-end model inference time with the batch size of 1 on the ImageNet dataset (image size: 224×224).

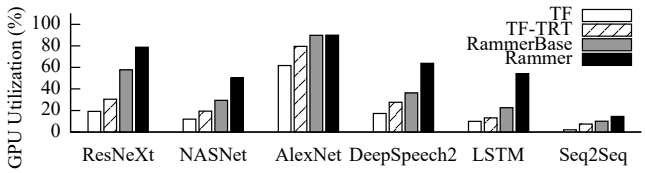


Figure 14: Comparison of GPU utilization.

Performance with larger input sizes. In our default settings, ResNeXt and NASNet are evaluated on images of 32×32 size in the CIFAR-10 dataset. To show RAMMER's performance on larger images, we also evaluate these two models on the ImageNet dataset with the same model hyper-parameters in their original papers [49, 54]. Specifically, ResNeXt on ImageNet uses 101 layers with cardinality of 64 and bottleneck width of 4d; and for NASNet, the number of repeated cells is 4 and the number of filters is 1056. Figure 13 shows the end-to-end model inference time. From the results, we observe that using larger input size has little impacts on RAMMER's performance gain. For example, using ImageNet, RAMMER can still outperform TF by 18.91×, TVM by 4.96×, and even TF-TRT by 2.06× on ResNeXt. For the NASNet model, RAMMER can also get 6.99×, 1.33× and 2.34× performance gains compared with TF, TVM and TF-TRT respectively. The significant performance improvement is mainly because that the model structure for larger dataset usually have more inter-operator parallelism that can be better leveraged by RAMMER's optimization. For example, the cardinality for ResNeXt is increased from 16 to 64 when replacing the dataset from CIFAR-10 to ImageNet.

Note that in the above evaluations, RAMMERBASE can already get a comparable or even better performance than compilers like TF-XLA and TVM. Thus, we will use RAMMERBASE as the baseline of the state-of-the-art compiler and TF-TRT as the state-of-the-art DNN inference library to evaluate the benefits of RAMMER in the rest of the evaluations. RAMMERBASE can also help remove the side effects caused by different implementations in the performance comparison.

5.2.2 GPU Utilization

RAMMER's scheduling enables *r*Tasks from different operators to execute alongside each other to achieve better GPU utilization. We evaluate the utilization improvement by RAMMER through comparing it with both TF, TF-TRT and RAMMERBASE. Figure 14 shows the average utilization for the 6 DNN models (with batch size of 1) through their execution time. The average GPU utilization only accounts for kernel execution, excluding other stages like operator emitting. Specifically, we use the metric *SM-efficiency* provided by NVIDIA profiler `nvprof` [6] to measure the utilization, which calculates the percentage of time when at least one warp is active on a multiprocessor. Compared to TF and TF-TRT, RAMMER can improve GPU utilization by 4.32× and 2.45× on average respectively across different models. This improvement comes from both the lower runtime scheduling overhead and the capability to co-schedule operators in RAMMER. Through comparing RAMMER with highly optimized RAMMERBASE, which uses the same set of kernels, our evaluation shows that RAMMER's scheduling by itself can improve the utilization by 1.61× on average, and up to 2.39× for the LSTM-TC model.

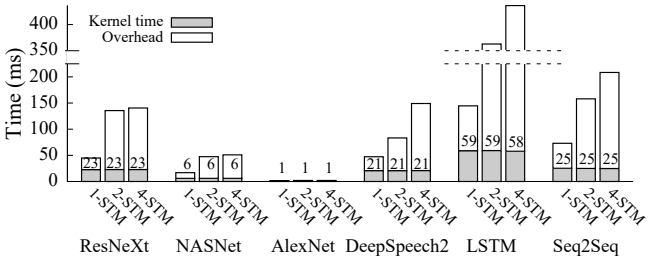


Figure 15: TF performance with different stream(STM) number. (Note: The number atop a bar indicates kernel time.)

As mentioned in §2, modern GPUs support the multi-streaming mechanism to increase utilization through concurrently scheduling independent kernels. We evaluate the efficiency of multi-streaming by increasing the stream numbers in TF. Figure 15 shows both the end-to-end execution time and the kernel time when using stream number of 1, 2, and 4 for each model. We observe that using more streams can harm the end-to-end performance, a phenomenon observed by others [45]. For example, using 4 streams increases the end-to-end time by 2.72× on average compared with using a single stream. Moreover, the kernel time in each model only sees very small reduction after enabling multi-streaming, which implies most kernels are still sequentially executed, thus providing little improvement on the GPU utilization. The major reason is because multi-streaming introduces even higher operator scheduling overhead, as shown in Figure 15.

5.2.3 Scheduling Overhead

The techniques proposed by RAMMER can effectively reduce scheduling overhead. To verify this, we evaluate the run-time

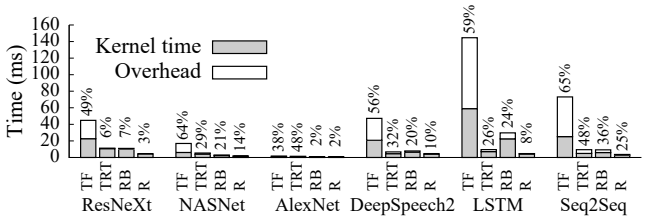


Figure 16: GPU scheduling overhead on different models. (The number atop a bar indicates the overhead in percentage.) TF: TensorFlow, TRT: TF-TRT, RB: RAMMERBASE, R: RAMMER

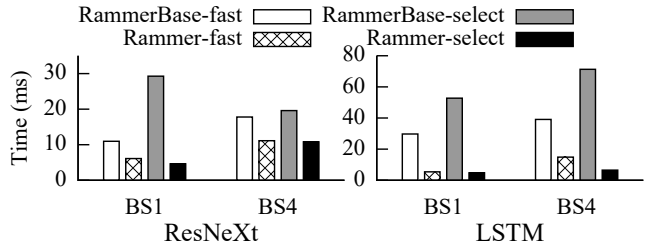


Figure 17: Performance with different kernel sets and batch sizes(BS).

scheduling overhead by comparing RAMMER with both TF, TF-TRT and RAMMERBASE. Figure 16 shows the total kernel time and the scheduling overhead (i.e., the time not spent on actual computation) for each model. Specifically, compared with TF, RAMMERBASE can reduce the scheduling time from an average of 32.29 milliseconds to only 2.27 milliseconds (overhead percentage from 55.41% to 18.43%) over all models. Even compared with TF-TRT, RAMMERBASE can reduce the average scheduling overhead from 31.38% to 18.43%. RAMMERBASE achieves this reduction by optimizing the scheduling execution code path and leveraging operator fusion to reduce kernel launches. The significant reduction demonstrates the heavy overhead of operator scheduling in existing DNN frameworks. Compared with RAMMERBASE, RAMMER can further reduce the average overhead from 2.27 milliseconds to 0.37 milliseconds, a 6.14× reduction. This significant reduction is due to static compile-time operator scheduling, i.e. packing operators into *r*Program so that several operators can be executed by a single GPU kernel launch.

5.2.4 Interplay of Intra and Inter Operator Scheduling

RAMMER enables scheduling policies to optimize the interplay of intra and inter operator scheduling, instead of just focusing on making individual operators fast. This is implemented through selecting appropriate *r*Kernel for each *r*Operator, as introduced in §3.3. We evaluate the effect of such scheduling by using two sets of kernels: the fastest ker-

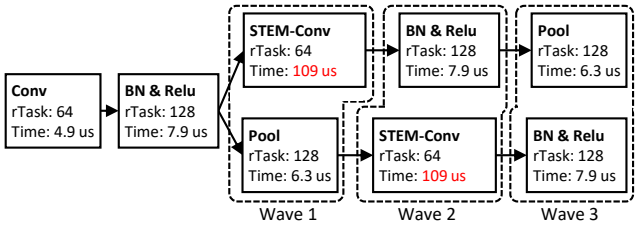


Figure 18: An irregular DFG generated by NASBench

nels only for each individual operator, and the kernels selected by RAMMER's scheduling policy. Figure 17 shows the performance of RAMMER and RAMMERBASE with these two kernel sets on two representative CNN and RNN models, i.e., ResNeXt and LSTM-TC. First, no matter which set of kernels is used, RAMMER can always improve the performance significantly. For example, if RAMMER uses the same fastest kernels (i.e., the RAMMER-fast) as used in RAMMERBASE (i.e., RAMMERBASE-fast), it can improve the performance by 2.89× on average. If more *r*Kernels are available for a given *r*Operator and RAMMER can select kernels based on its policy (i.e., the RAMMER-select), it can further improve the end-to-end performance by 1.44× on average, and up to 2.28× compared with RAMMER-fast, even though the selected kernels may be not the fastest in isolation. In fact, if we use these kernels in RAMMERBASE (i.e., the RAMMERBASE-select), its performance will drop by 1.84× on average.

We further perform detailed analysis of the kernels used in LSTM-TC model with batch size of 4. For example, for the `Matmul` operator, the fastest kernel uses 1,024 *r*Tasks to get the optimal execution time of 4.28 microseconds; while the selected kernel by RAMMER only consists of 16 *r*Tasks and gets a slower execution time of 7.46 microseconds when launched alone. However, RAMMER chooses this kernel to trade a slower individual kernel (by reducing intra-operator parallelism) for a better overall performance (through increasing the inter-operator parallelism), thanks to the holistic scheduling capability of RAMMER.

5.2.5 Fine-grained Synchronization

As a synchronization mechanism, barrier-*r*Task provides some extra optimization spaces for the DFGs with irregular structure, which is common in the models generated by neural architecture search (NAS) [54]. To highlight such extra benefit, we leverage NASBench [50], a state-of-the-art NAS benchmark, to randomly generate 5,000 modules, where each module is a small DFG that consists of up to 9 operators and 7 edges. We first compare the end-to-end performance of RAMMER and RAMMERBASE on all these modules, which shows RAMMER can improve the performance by 1.28× on average, and up to 3.40× than RAMMERBASE. Among all these modules, our measurement shows that 28.3% of them has obvious irregular structures, e.g., heterogeneous operators

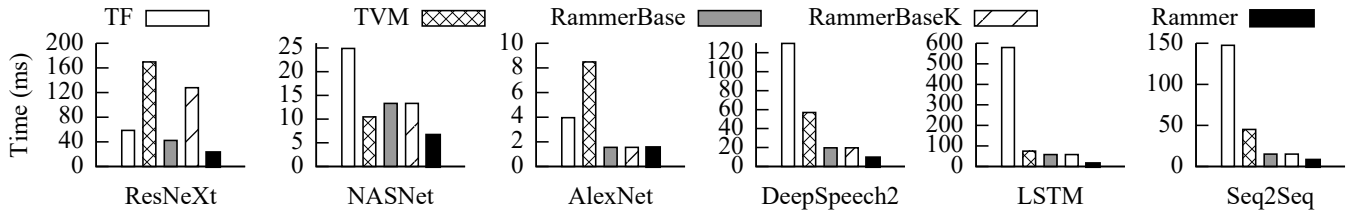


Figure 19: End-to-end model inference time with batch size of 1 on AMD MI50 GPU.

in a wave decided by our policy (Algorithm 1). For these modules, we compare the end-to-end performance of using our barrier-*r*Task implementation and a global barrier. The results show that using the barrier-*r*Task can provide extra performance speedup of 1.11× on average and up to 1.89×. Figure 18 illustrates one of such modules, where the execution time and *r*Task number in each operator are also listed. For such a DFG, our barrier-*r*Task provides a possibility to overlap the execution of operators from different waves (e.g., the two `STEM-Conv` operators from wave 1 and 2) through removing the global barriers between waves and inserting fine-grained *r*Task-level synchronizations.

5.3 Evaluation on Other Accelerators

5.3.1 End-to-end Performance on ROCm GPUs

We evaluate the efficiency of RAMMER on AMD ROCm GPUs by comparing it with TF, TVM, and RAMMERBASE. TF-XLA is not included because it cannot be successfully enabled on AMD GPUs in our experiments, and TensorRT is not included because it is proprietary and is exclusive for NVIDIA. Figure 19 shows the end-to-end performance of the 6 benchmarks with batch size of 1. Compared with TF, RAMMER can outperform it by 13.95× on average, and up to 41.14× for the LSTM-TC model. Compared to TVM, RAMMER can improve the performance by 5.36× on average, and up to 7.57×. Note that we fail to make the TVM auto tuning feature works on ROCm GPUs, so TVM just uses its default kernels in this experiment. Compared with RAMMERBASE, we can see that the proposed scheduling of RAMMER's can bring average of 2.19× and up to 4.12× speedup. Finally, RAMMERBASEK in the figures is exactly the same as RAMMERBASE, except that it uses kernels from RAMMER. Notice that RAMMER might not always choose the fastest kernel implementations for the *r*Operators. Though there are little performance change for most models, for the ResNeXt model there is a 3.02× performance drop. This demonstrates the importance of the interplay of scheduling and kernel selection.

5.3.2 End-to-end Performance on Graphcore IPU

We also conduct a preliminary evaluation of RAMMER on a Graphcore IPU. In this experiment, we choose only the three

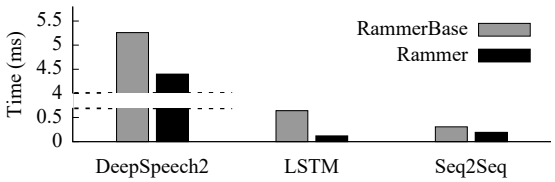


Figure 20: End-to-end model inference time with batch size of 1 on Graphcore IPU.

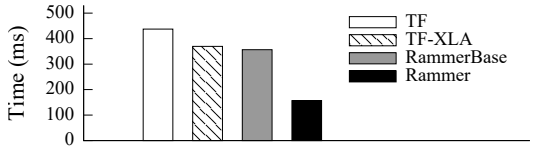


Figure 21: End-to-end model training time of LSTM-TC with batch size of 256 on NVIDIA V100 GPU. Note that TVM and TF-TRT do not support training, hence the data is missing.

RNN benchmarks, again, because it takes effort implementing efficient *r*Operators to support other models. Currently, RAMMER only supports a single IPU device. We leave the multi-IPU support of RAMMER to future work. For the three RNN models, due to the limited memory available on IPU (256 KB on each tile), we configure the layers of these models to 4 in order to fit in a single IPU. Figure 20 shows the end-to-end performance of RAMMER on these models with batch size of 1. It shows that RAMMER's preliminary implementation can bring up to 5.37× performance improvement compared with RAMMERBASE, which demonstrates the applicability and effectiveness of the abstractions of RAMMER on new accelerator architectures.

6 Discussion

Having shown the advantages, we discuss some RAMMER's limitations and future work in this section.

Performance gain on large batch sizes. RAMMER's benefits are more significant when the intra-operator parallelism

is insufficient to saturate hardware. This is the case when the input batch size is small, often found in online DNN inference. Moreover, our preliminary experiment shows that this is also the case for some model training workloads with large batch sizes (e.g., 256), such as the LSTM-TC model. Figure 21 shows the training performance of LSTM-TC with batch size of 256. As it shows, with holistic optimizations on both intra- and inter-operator parallelism, RAMMER improves the performance by 2.28× than our baseline implementation RAMMERBASE and 2.36× than TF-XLA. We will leave a more detailed analysis and further optimizations on model training with large batch sizes as our future work.

Dynamic graph. Currently, RAMMER only supports static graph. For DFGs with dynamic control flow [51], RAMMER can compile each of the static sub-graphs, e.g., a branch of conditionals or a body of loops, into individual rPrograms. We leave this implementation to our future work.

Inter-job scheduling. RAMMER focuses on optimizing a single deep learning job and is orthogonal to inter-job scheduling, e.g., through scheduling multiple models in a batch or precisely controlling each job's hardware resource with vDevice. Nevertheless, it is an interesting topic to explore the possibility to co-schedule *r*Tasks not only from different operators, but from different jobs within an accelerator.

7 Related Work

DNN compiler optimization can be generally divided into two classes based on its two-layered representations. DFG-level optimizations, such as operator fusion, are exploited in many DNN frameworks and compilers, e.g., TensorFlow [18], PyTorch [15], TVM [23], XLA [17], etc. TASO [34] proposes an automatic graph substitutions approach to optimize the DFG. On the operator-level, recent work has leveraged different approaches to tune and generate efficient hardware-specific operator code, e.g., AutoTVM [24], Tensor comprehension [47], FlexTensor [53], Tiramisu [21], Halide [43], etc. RAMMER is compatible with all these optimizations through taking an optimized DFG as input and generating efficient *r*Kernels with those kernel generators.

DNN inference and its optimization have attracted a lot of recent attention. DeepCPU [52], BatchMaker [27], GRNN [32], and NeoCPU [40] optimize the inference for RNN or CNN specific models on either CPU or GPUs. Jain et al. [33] proposes to leverage both temporal and spatial multiplexing for multiple inference jobs to improve the GPU utilization. RAMMER differentiates with these works in two aspects: 1) RAMMER can apply to general DNN models and accelerators; and 2) more than just compiler optimizations, RAMMER provides a new abstraction and a larger optimization space for DNN computation. Astra [45] exploits the predictability of DNN to perform online optimization for DNN training, while RAMMER leverages the same property to reduce the individual *r*Task scheduling overhead. There are also many inference systems proposed to optimize the overall throughout under the guaranteed query latency, e.g., Nexus [44], PRETZEL [38], Clipper [25], TF-serving [42], etc. RAMMER instead focuses on optimizing a single model and is orthogonal to these works.

Some other work from the GPU community has proposed software-based schedulers within a GPU to schedule general workload. For example, Juggler [22] proposes a framework to dynamically execute a job represented as a DAG of tasks. Wu et al. [48] proposes a software approach to control the job locality on SMs. However, driven by the property of DNN workload, RAMMER proposes a new computation representation with *r*Task and *r*Operator; and adopts a compile-time scheduling approach to avoid runtime overhead systemically.

8 Conclusion

DNN computation suffers from unnecessary overheads due to the fundamental limitations of existing deep learning frameworks, which adopt a two-layer scheduling design that manages the inter-operator scheduling in the framework and delegates intra-operator scheduling to the hardware accelerator. RAMMER addresses this issue with a holistic compiler solution that (1) provides an *r*Task-operator abstraction that exposes the fine-grained intra-operator parallelism. (2) virtualizes the modern accelerator with parallel execution units to expose the hardware's fine-grained scheduling capability. (3) leverages the predictability of DNN computation to transform run-time scheduling into a problem of generating compile-time *r*Task execution plans. Our evaluations show that RAMMER can achieve significant improvements compared to native deep learning frameworks, compilation frameworks and even vendor-specific inference engine on GPUs. This positions RAMMER as a new enhancement to the existing ecosystem of DNN compiler infrastructure.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof. Jinyang Li, for their extensive suggestions. We thank Jim Jernigan and Kendall Martin from the Microsoft Grand Central Resources team for the support of GPUs. Fan Yang thanks the late Pearl, his beloved cat, for her faithful companion during writing this paper. This work was partially supported by the National Natural Science Foundation of China under Grant No. 61972004.

References

[1] AMD ROCm Platform. https://github.com/RadeonOpenCompute/ROCm.

[2] CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[3] Cooperative Groups. https://devblogs.nvidia.com/cooperative-groups/.

[4] CUDA Driver API. http://docs.nvidia.com/cuda/cuda-driver-api.

[5] CUDA NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[6] CUDA nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/.

[7] Depthwise separable 2D convolution. https://www.tensorflow.org/api_docs/python/tf/keras/layers/SeparableConv2D.

[8] HIP Programming Guide. https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html.

[9] Intel MKL-DNN. https://github.com/oneapi-src/oneDNN.

[10] IPU PROGRAMMER'S GUIDE. https://www.graphcore.ai/docs/ipu-programmers-guide.

[11] LibriSpeech ASR corpus . http://www.openslr.org/12/.

[12] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[13] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[14] ONNX. https://onnx.ai/.

[15] PyTorch. https://pytorch.org/.

[16] TorchScript. https://pytorch.org/docs/stable/jit.html.

[17] XLA. https://www.tensorflow.org/xla.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association.

[19] Dario Amodei and Sundaram Ananthanarayanan et al. Deep speech 2 : End-to-end speech recognition in english and mandarin. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 173–182, New York, New York, USA, 20–22 Jun 2016. PMLR.

[20] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2018.

[21] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[22] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. Juggler: A dependence-aware task-based execution framework for gpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 54–67, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[24] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3389–3400. Curran Associates, Inc., 2018.

[25] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and*

Implementation (NSDI 17), pages 613–627, Boston, MA, 2017. USENIX Association.

[26] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[27] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys 18)*, 2018.

[28] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 12(1), March 2019.

[29] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *2012 Innovative Parallel Computing (InPar)*, pages 1–14, 2012.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[32] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 41. ACM, 2019.

[33] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. *CoRR*, abs/1901.00041, Dec 2018.

[34] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[36] Junjie Lai and Andre Seznec. Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, USA, 2013. IEEE Computer Society.

[37] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, USA, 2006.

[38] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.

[39] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., USA, 2004.

[40] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing cnn model inference on cpus. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1025–1040, USA, 2019. USENIX Association.

[41] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, November 2010.

[42] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[43] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[44] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 909–923, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[47] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[48] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 119–130, New York, NY, USA, 2015. ACM.

[49] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *arXiv preprint arXiv:1611.05431*, 2016.

[50] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: Towards reproducible neural architecture search. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7105–7114, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[51] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[52] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, Boston, MA, July 2018. USENIX Association.

[53] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery.

[54] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.



A Tensor Compiler for Unified Machine Learning Prediction Serving

Supun Nakandala[c,*], *Karla Saur*[m], *Gyeong-In Yu*[s,*], *Konstantinos Karanasos*[m], *Carlo Curino*[m], *Markus Weimer*[m], *Matteo Interlandi*[m]

[m]Microsoft, [c]UC San Diego, [s]Seoul National University

{<name>.<surname>}@microsoft.com, snakanda@eng.ucsd.edu, gyeongin@snu.ac.kr

Abstract

Machine Learning (ML) adoption in the enterprise requires simpler and more efficient software infrastructure—the bespoke solutions typical in large web companies are simply untenable. Model scoring, the process of obtaining predictions from a trained model over new data, is a primary contributor to infrastructure complexity and cost as models are trained once but used many times. In this paper we propose HUMMINGBIRD, a novel approach to model scoring, which compiles featurization operators and traditional ML models (e.g., decision trees) into a small set of tensor operations. This approach inherently reduces infrastructure complexity and directly leverages existing investments in Neural Network compilers and runtimes to generate efficient computations for both CPU and hardware accelerators. Our performance results are intriguing: despite replacing imperative computations (e.g., tree traversals) with tensor computation abstractions, HUMMINGBIRD is competitive and often outperforms hand-crafted kernels on micro-benchmarks on both CPU and GPU, while enabling seamless end-to-end acceleration of ML pipelines. We have released HUMMINGBIRD as open source.

1 Introduction

Enterprises increasingly look to Machine Learning (ML) to help solve business challenges that escape imperative programming and analytical querying [35]—examples include predictive maintenance, customer churn prediction, and supply-chain optimizations [46]. To do so, they typically turn to technologies now broadly referred to as *"traditional ML"*, to contrast them with Deep Neural Networks (DNNs). A recent analysis by Amazon Web Services found that 50 to 95% of all ML applications in an organization are based on traditional ML [38]. An analysis of 6*M* notebooks in public GitHub repositories [64] paints a similar picture: NumPy [69], Matplotlib [11], Pandas [7], and scikit-learn [62] are the four most used libraries—all four provide functions for traditional ML. As a point of comparison with DNN frameworks, scikit-learn is used about 5 times more than PyTorch [61] and TensorFlow [13] combined, and growing faster than both. Acknowledging this trend, traditional ML capabilities have been recently added to DNN frameworks, such as the ONNX-ML [4] flavor in ONNX [25] and TensorFlow's TFX [39].

When it comes to owning and operating ML solutions, enterprises differ from early adopters in their focus on long-term costs of ownership and amortized return on investments [68]. As such, enterprises are highly sensitive to: (1) complexity, (2) performance, and (3) overall operational efficiency of their software infrastructure [14]. In this work we focus on *model scoring* (i.e., the process of getting a prediction from a trained model by presenting it with new data), as it is a key driving factor in each of these regards. First, each model is trained once but used multiple times for scoring in a variety of environments, thus scoring dominates infrastructure complexity for deployment, maintainability, and monitoring. Second, model scoring is often in the critical path of interactive and analytical enterprise applications, hence its performance (in terms of latency and throughput) is an important concern for enterprises. Finally, *model scoring is responsible for 45-65% of the total cost of ownership of data science solutions* [38].

Predictive Pipelines. The output of the iterative process of designing and training traditional ML models is *not just* a model but a *predictive pipeline*: a Directed Acyclic Graph (DAG) of operators. Such pipelines are typically comprised of up to tens of operators out of a set of hundreds [64] that fall into two main categories: (1) *featurizers*, which could be either stateless imperative code (e.g., string tokenization) or data transformations fit to the data (e.g., normalization); and (2) *models*, commonly decision tree ensembles or (generalized) linear models, fit to the data. Note that the whole pipeline is required to perform a prediction.

A Missing Abstraction. Today's featurizers and model implementations *are not expressed in a shared logical abstraction, but rather in an ad-hoc fashion* using programming languages such as R, Python, Java, C++, or C#. This hints to the core problem with today's approaches to model scoring: *the combinatorial explosion of supporting many operators*

*The work was done while the author was at Microsoft.

(and frameworks) across multiple target environments. Figure 1 (top) highlights this visually by showing how existing solutions lead to an $O(N \times M)$ explosion to support N operators from various ML frameworks against M deployment environments (e.g., how to run a scikit-learn model on an embedded device?). Furthermore, [64] shows that the number of libraries used in data science (a metric correlated to N) increased by roughly $4\times$ in the last 2 years. Our expectation is that M is also destined to grow as ML is applied more widely across a broad range of enterprise applications and hardware (e.g., [1, 15, 30, 48, 49]). From the vantage point of implementing runtimes for model scoring, this is a daunting proposition. We argue that any brute-force approach directly tackling all combinations would dilute engineering focus leading to costly and less optimized solutions. In fact, today, with very few exceptions (e.g., NVIDIA RAPIDS [3] for GPU), traditional ML operators are only implemented for CPUs.

This state of affairs is in contrast with the DNN space, where neural networks are authored using tensor transformations (e.g., multiplications, convolutions), providing an algebraic abstraction over computations. Using such abstractions rather than imperative code not only enables evolved optimizations [33, 41] but also facilitates support for diverse environments (such as mobile devices [26], web browsers [32], and hardware accelerators [15, 48, 49]), unlocking new levels of performance and portability.

Our Solution. To bypass this $N \times M$ explosion in implementing traditional ML operators, we built HUMMINGBIRD (HB for short). HB leverages compilation and optimization techniques to translate a broad set of traditional ML operators into a small set of K core operators, thereby reducing the cost to $O(N) + O(K \times M)$, as shown in Figure 1 (bottom). This is also the key intuition behind the ONNX model format [25] and its various runtimes [6]. However, with HB we take one further bold step: we demonstrate that this set of core operators can be reduced to tensor computations and therefore be executed over DNN frameworks. This allows us to piggyback on existing investments in DNN compilers, runtimes, and specialized-hardware, and reduce the challenge of "running K operators across M environments" for traditional ML to just $O(N)$ operator translations. This leads to improved performance and portability, and reduced infrastructure complexity.

Contributions. In this paper we answer three main questions:

1. Can traditional ML operators (both linear algebra-based such as linear models, and algorithmic ones such as decision trees) be translated to tensor computations?
2. Can the resulting computations in tensor space be competitive with the imperative alternatives we get as input (e.g., traversing a tree)?
3. Can HB help in reducing software complexity and improving model portability?

Concretely, we: (1) port thousands of benchmark predictive pipelines to two DNN backends (PyTorch and TVM);

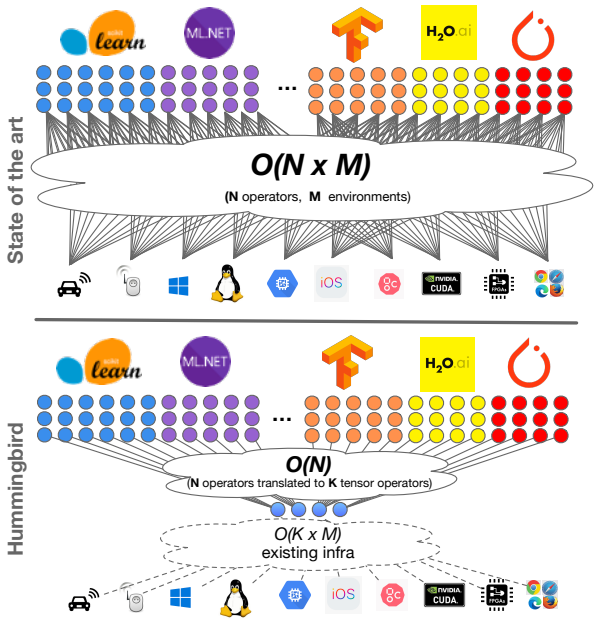


Figure 1: Prediction serving complexity: state-of-the-art (top) vs. HUMMINGBIRD (bottom).

(2) show that we can seamlessly leverage hardware accelerators and deliver speedups of up to $3\times$ against hand-crafted GPU kernels, and up to $1200\times$ for predictive pipelines against state-of-the-art frameworks; and (3) qualitatively confirm improvements in software complexity and portability by enabling scikit-learn pipelines to run across CPUs and GPUs.

HB is open source under the MIT license [1], and is part of the PyTorch ecosystem [28]. We are integrating HB with other systems, such as the ONNX converters [58].

Organization. The remainder of the paper is organized as follows. Section 2 provides some background, and Section 3 presents an overview of HB. Section 4 describes the compilation from traditional ML to tensor computations, whereas Section 5 discusses various optimizations. Section 6 presents our evaluation. Section 7 is related work, then we conclude.

2 Background and Challenges

We first provide background on traditional ML and DNNs. We then explain the challenges of compiling traditional ML operators and predictive pipelines into tensor computations.

2.1 Traditional ML and DNNs

Traditional Predictive Pipelines. The result of the data science workflow over traditional ML are predictive pipelines, i.e., DAG of operators such as trained models, preprocessors, featurizers, and missing-value imputers. The process of presenting a trained predictive pipeline with new data to obtain a prediction is referred to in literature interchangeably as: model scoring/inference/serving, pipeline evaluation, or prediction serving. We favor model scoring in our writing.

[1] https://github.com/microsoft/hummingbird

Packaging a trained pipeline into a single artifact is common practice [36]. These artifacts are then embedded inside host applications or containerized and deployed in the cloud to perform model scoring [43, 63]. ML.NET [36] (.NET-based), scikit-learn [62] (Python-based), and H_2O [9] (Java-based) are popular toolkits to generate pipelines. However, they are primarily optimized for training. Scoring predictive pipelines is challenging, as their operators are implemented in imperative code and do not follow a shared abstraction. Supporting every operator in all target environments requires a huge effort, which is why these frameworks have limited portability.

DNNs. Deep Neural Networks (DNNs) are a family of ML models that are based on artificial neurons [47]. They take raw features as input and perform a series of transformation operations. Unlike traditional ML, transformations in DNNs are drawn from a common abstraction based on tensor operators (e.g., generic matrix multiplication, element-wise operations). In recent years, DNNs have been extremely successful in vision and natural language processing tasks [45, 54]. Common frameworks used to author and train DNNs are TensorFlow [13], PyTorch [61], CNTK [10], and MXNet [12]. While these frameworks can also be used to perform model scoring, next we discuss systems specifically designed for that.

Runtimes for DNN Model Scoring. To cater to the demand for DNN model inference, a new class of systems has emerged. ONNX Runtime (ORT) [5] and TVM [41] are popular examples of such systems. These capitalize on the relative simplicity of neural networks: they accept a DAG of tensor operations as input, which they execute by implementing a small set of highly optimized operator kernels on multiple hardwares. Focusing on just the prediction serving scenario also enables these systems to perform additional inference-specific optimizations, which are not applicable for training. HB is currently compatible with all such systems.

2.2 Challenges

HB combines the strength of traditional ML pipelines on structured data [56] with the computational and operational simplicity of DNN runtimes for model scoring. To do so, it relies on a simple yet key observation: once a model is trained, it can be represented as a *prediction function* transforming input features into a prediction score (e.g., 0 or 1 for binary classification), regardless of the training algorithm used. The same observation naturally applies to featurizers fit to the data. Therefore, HB only needs to compile the prediction functions (not the training logic) for each operator in a pipeline into tensor computations and stitch them appropriately. Towards this goal, we identify two challenges.

Challenge 1: *How can we map traditional predictive pipelines into tensor computations?* Pipelines are generally composed of operators (with predictive functions) of two classes: *algebraic* (e.g., scalers or linear models) and *algorithmic* (e.g., one-hot encoder and tree-based models). While translating algebraic operators into tensor computations is straightforward, the key challenge for HB is the translation of algorithmic operators. Algorithmic operators perform arbitrary *data accesses and control flow decisions*. For example, in a decision tree ensemble potentially every tree is different from each other, not only with respect to the structure, but also the decision variables and the threshold values. Conversely, tensor operators perform *bulk operations* over the entire set of input elements.

Challenge 2: *How can we achieve efficient execution for tensor-compiled traditional ML operators?* The ability to compile predictive pipelines into DAGs of tensor operations does not imply adequate performance of the resulting DAGs. In fact, common wisdom would suggest the opposite: even though tensor runtimes naturally support execution on hardware accelerators, tree-based methods and commonly used data transformations are well known to be difficult to accelerate [42], even using custom-developed implementations.

3 System Overview

In this section we explain our approach to overcome the challenges outlined in Section 2.2, and present HB's architecture and implementation details. We conclude this section by explaining assumptions and limitations.

3.1 High-level Approach

In HB, we cast algorithmic operators into tensor computations. You will notice that this transformation *introduces redundancies*, both in terms of *computation* (we perform more computations than the original traditional ML operators) and *storage* (we create data structures that store more than what we actually need). Although these redundancies might sound counter-intuitive at first, we are able to transform the arbitrary data accesses and control flow of the original operators into tensor operations that lead to efficient computations by leveraging state-of-the-art DNN runtimes.

For a given traditional ML operator, there exist different strategies for compiling it to tensor computations, each introducing a different degree of redundancy. We discuss such strategies for representative operators in Section 4. The optimal tensor implementation to be used varies and is informed by model characteristics (e.g., tree-structure for tree-based models, or sparsity for linear models) and runtime statistics (e.g., batch size of the inputs). *Heuristics at the operator level*, *runtime-independent optimizations at the pipeline level*, and *runtime-specific optimizations at the execution level* enable HB to further improve predictive pipelines performance end-to-end. The dichotomy between runtime-independent and runtime-specific optimizations allow us to both (1) apply optimizations unique to traditional ML and not captured by the DNN runtimes; and (2) exploit DNN runtime optimizations once the traditional ML is lowered into tensor computations. Finally, HB is able to run end-to-end pipelines on the hardware platforms supported by the target DNN runtimes.

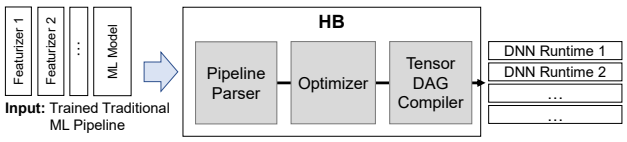


Figure 2: High-level architecture of HB.

3.2 System Architecture and Implementation

The high-level architecture of HB is shown in Figure 2. HB has three main components: (1) *Pipeline Parser*, (2) *Optimizer*, and (3) *Tensor DAG Compiler*.

Pipeline Parser. In this phase, input pipelines are parsed one operator at a time, and each operator is *wrapped* into a *container* object. Each operator's container maintains (1) the inputs and outputs of the operator, and (2) the *operator signature* that codifies the operator type (e.g., "scikit-learn decision tree"). HB parser also introduces a set of *extractor functions* that are used to extract the parameters of each operator (e.g., weights of a linear regression, thresholds of a decision tree). Operator signatures dictate which extractor function should be used for each operator. At startup time, extractor functions are registered into a hash table, mapping operator signatures to the related extractor function. HB parser is extensible, allowing users to easily add new extractor functions. HB currently supports over 40 scikit-learn operators (listed in Table 1), as well as parsers for XGBoost [40], LightGBM [51], and ONNX-ML [4]. At the end of the parsing phase, the input pipeline is "logically" represented in HB as a DAG of containers storing all the information required for the successive phases. HB parser is based on skl2onnx [31].

Optimizer. In this phase, the DAG of containers generated in the parsing phase is traversed in topological order in two passes. During the first traversal pass, the Optimizer extracts the parameters of each operator via the referenced extractor function and stores them in the container. Furthermore, since HB supports different operator implementations based on the extracted parameters, the Optimizer annotates the container with the compilation strategy to be used for that specific operator (5.1). During the second pass, HB tries to apply runtime-independent optimizations (5.2) over the DAG.

Tensor DAG Compiler. In this last phase, the DAG of containers is again traversed in topological order and a *conversion-to-tensors function* is triggered based on each operator signatures. Each conversion function receives as input the extracted parameters and generates a PyTorch's *neural network module* composed of a small set of tensor operators (listed in Table 2). The generated module is then exported into the target runtime format. The current version of HB supports PyTorch/TorchScript, ONNX, and TVM output formats. The runtime-specific optimizations are triggered at this level.

Table 2: PyTorch tensor operators used by the Tensor DAG Compiler.

`matmul, add, mul, div, lt, le, eq, gt, ge, &, |, ≪, ≫, bitwise_xor, gather, index_select, cat, reshape, cast, abs, pow, exp, arxmax, max, sum, relu, tanh, sigmoid, logsumexp, isnan, where`

Table 1: Scikit-learn operators currently supported in HB.

Supported ML Models
LogisticRegression, SVC, NuSVC, LinearSVC, SGDClassifier, LogisticRegressionCV, DecisionTreeClassifier/Regression, RandomForestClassifier/Regression, ExtraTreesClassifier/Regressor, GradientBoostingClassifier/Regression, HistGradientBoostingClassifier/Regressor, IsoltationForest, MLPClassifier, BernoulliNB, GaussianNB, MultinomialNB
Supported Featurizers
SelectKBest, VarianceThreshold, SelectPercentile, PCA, KernelPCA, TruncatedSVD, FastICA, SimpleImputer, Imputer, MissingIndicator, RobustScaler, MaxAbsScaler, MinMaxScaler, StandardScaler, Binarizer, KBinsDiscretizer, Normalizer, PolynomialFeatures, OneHotEncoder, LabelEncoder, FeatureHasher

3.3 Assumptions and Limitations

In this paper, we make a few simplifying assumptions. First, we assume that predictive pipelines are "pure", i.e., they do not contain arbitrary user-defined operators. There has been recent work [65] on compiling imperative UDFs (user-defined functions) into relational algebra, and we plan to make use of such techniques in HB in the future. Second, we do not support sparse data well. We found that current support for sparse computations on DNN runtimes is primitive and not well optimized. We expect advances in DNN frameworks to improve on this aspect—TACO [52] is a notable such example. Third, although we support string operators, we currently do not support text feature extraction (e.g., `TfidfVectorizer`). The problem in this case is twofold: (1) compiling regex-based tokenizers into tensor computations is not trivial, and (2) representing arbitrarily long text documents in tensors is still an open challenge. Finally, HB is currently limited by single GPU memory execution. Given that several DNN runtimes nowadays support distributed processing [57, 66], we plan to investigate distributed inference as future work.

4 Compilation

HB supports compiling several algorithmic operators into tensor computations. Given their popularity [64], in Section 4.1 we explain our approach for tree-based models. Section 4.2 gives a summary of other techniques that we use for both algorithmic and arithmetic operators.

4.1 Compiling Tree-based Models

HB has three different strategies for compiling tree-based models. Strategies differ based on the degree of redundancy introduced. Table 3 explains the notation used in this section. We summarize the worst-case runtime and memory footprints of each strategy in Table 4. HB currently supports only trees built over numerical values: support for missing and categorical values is under development. For the sake of presentation, we assume all decision nodes perform $<$ comparisons.

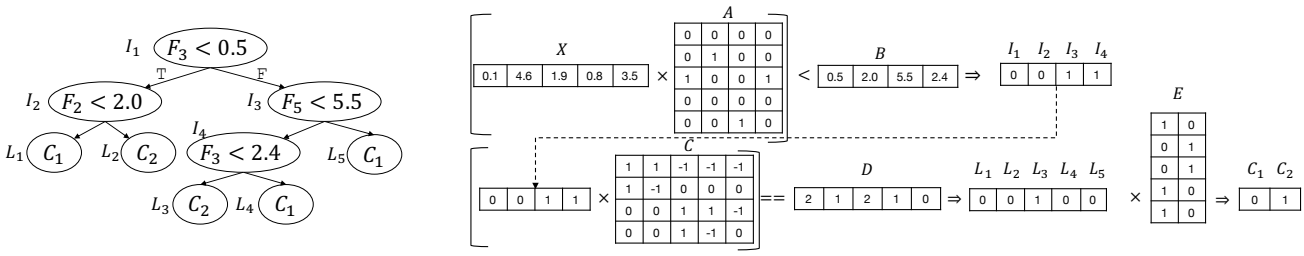


Figure 3: Compiling an example decision tree using the GEMM strategy.

Table 3: Notation used in Section 4.1

Symbol	Description
N, I, L, F, C	Ordered lists with all nodes, internal nodes, leaf nodes, features, and classes, respectively.
$X \in \mathbb{R}^{n \times \lvert F \rvert}$	Input records (n is the number of records).
$A \in \mathbb{R}^{\lvert F \rvert \times \lvert I \rvert}$	$A_{i,j} = \begin{cases} 1, & I_j \text{ evaluates } F_i \\ 0, & \text{Otherwise} \end{cases}$
$B \in \mathbb{R}^{\lvert I \rvert}$	$B_i = \texttt{ThresholdValue}(I_i)$
$C \in \mathbb{R}^{\lvert I \rvert \times \lvert L \rvert}$	$C_{i,j} = \begin{cases} -1, & L_j \in \texttt{RightSubTree}(I_i) \\ 1, & L_j \in \texttt{LeftSubTree}(I_i) \\ 0, & \text{Otherwise} \end{cases}$
$D \in \mathbb{R}^{\lvert L \rvert}$	$D_k = \sum_{k \in L \xrightarrow{path} \texttt{Root}} \mathbf{1}(k == \texttt{LeftChild}(\texttt{Parent}(k)))$
$E \in \mathbb{R}^{\lvert L \rvert \times \lvert C \rvert}$	$E_{i,j} = \begin{cases} 1, & L_i \xrightarrow{map\ to} C_j \\ 0, & \text{Otherwise} \end{cases}$

Table 4: Worst-case memory and runtime analysis of different tree compilation strategies, assuming the number of input records and number of trees are fixed. The notation is explained in Table 3.

Strategy	Memory	Runtime
GEMM	$O(\lvert F \rvert \lvert N \rvert + \lvert N \rvert^2 + \lvert C \rvert \lvert N \rvert)$	$O(\lvert F \rvert \lvert N \rvert + \lvert N \rvert^2 + \lvert C \rvert \lvert N \rvert)$
TT	$O(\lvert N \rvert)$	$O(\lvert N \rvert)$
PTT	$O(2^{\lvert N \rvert})$	$O(\lvert N \rvert)$

Strategy 1: GEMM. We cast the evaluation of a tree as a series of three GEneric Matrix Multiplication (GEMM) operations interleaved by two element-wise logical operations. Given a tree, we create five tensors which collectively capture the tree structure: A, B, C, D, and E. A captures the relationship between input features and internal nodes. B is set to the threshold value of each internal node. For any leaf node and internal node pair, C captures whether the internal node is a parent of that internal node, and if so, whether it is in the left or right sub-tree. D captures the count of the internal nodes in the path from a leaf node to the tree root, for which the internal node is the left child of its parent. Finally, E captures the mapping between leaf nodes and the class labels. Given these tensors, Algorithm 1 presents how we perform tree scoring for a batch of input records X. A graphical representation of an execution of the GEMM strategy is depicted in Figure 3.

The first GEMM is used to match each input feature with the internal node(s) using it. The following $<$ operations is used to evaluate all the internal decision nodes and produces a tensor of 0s and 1s based on the false/true outcome of the conditions. The second GEMM operation generates an encoding for the path composed by the true internal nodes, while the successive $==$ operation returns the leaf node selected by the encoded path. Finally, the third GEMM operation maps the selected leaf node to the class label.

Algorithm 1 GEMM Strategy (Notation explained in Table 3)

Input : $X \in \mathbb{R}^{n \times \lvert F \rvert}$, Input records
Output : $R \in \{0,1\}^{n \times \lvert C \rvert}$, Predicted class labels

```
/* Evaluate all internal nodes                                   */
T ← GEMM(X, A)                          // T ∈ R^{n×|I|}
T ← T < B                               // T ∈ R^{n×|I|}

/* Find the leaf node which gets selected                        */
T ← GEMM(T, C)                          // T ∈ R^{n×|L|}
T ← T == D                              // T ∈ R^{n×|L|}

/* Map selected leaf node to class label                         */
R ← GEMM(T, E)                          // R ∈ R^{n×|C|}
```

This strategy can be easily applied to support tree ensembles and regression tasks too. For tree ensembles, we create the above 2-dimensional tensors for each tree and batch them together. As the number of leaf nodes and internal nodes can vary among trees, we pick the maximum number of leaf nodes and internal nodes for any tree as the tensor dimensions and pad the smaller tensor slices with zeros. During scoring, we invoke the batched variants of GEMM and logical operations and perform a final ReduceMean operation over the batched dimension to generate the ensemble output. For regression tasks, we initialize E with label values.

Strategy 2: TreeTraversal (TT). In the GEMM strategy, we incorporated a high degree of computational redundancy by evaluating all internal nodes and leaf nodes. Here, we try to reduce the computational redundancy by mimicking the typical tree traversal—but implemented using tensor operations. In this strategy, the tree structure is captured by five tensors: N_L, N_R, N_F, N_T, and N_C. We formally define these tensors in Table 5. The same column index (last dimension) across all tensors corresponds to the same tree node. N_L and N_R capture the indices of the left and right nodes for a given node. If the node is a leaf node, we set these to the index of the given node. Similarly, N_F and N_T capture the feature index and threshold

Table 5: Additional notation used in Strategy 2: TreeTraversal

Symbol	Description
$N_L \in \mathbb{Z}^{\lvert N\rvert}$	$N_{L_i} = \begin{cases} \texttt{LeftChild}(N_i), N_i \in I \\ i, \text{Otherwise} \end{cases}$
$N_R \in \mathbb{Z}^{\lvert N\rvert}$	$N_{R_i} = \begin{cases} \texttt{RightChild}(N_i), N_i \in I \\ i, \text{Otherwise} \end{cases}$
$N_F \in \mathbb{Z}^{\lvert N\rvert}$	$N_{F_i} = \begin{cases} k, (N_i \in I) \wedge (N_i \text{ evaluates } F_k) \\ 1, \text{Otherwise} \end{cases}$
$N_T \in \mathbb{R}^{\lvert N\rvert}$	$N_{T_i} = \begin{cases} \texttt{ThresholdValue}(N_i), N_i \in I \\ 0, \text{Otherwise} \end{cases}$
$N_C \in \mathbb{Z}^{\lvert N\rvert \times \lvert C\rvert}$	$N_{C_{i,k}} = \begin{cases} 1, (N_i \in L) \wedge (N_i \xrightarrow{map\ to} C_k) \\ 0, \text{Otherwise} \end{cases}$

Algorithm 2 TreeTraversal Strategy (Notation in Tables 5)

Input : $X \in \mathbb{R}^{n \times \lvert F\rvert}$, Input records
Output : $R \in \{0,1\}^{n \times \lvert C\rvert}$, Predicted class labels

```
/* Initialize all records to point to k, with k the index
   of Root node.                                          */
T_I ← {k}^n                                   // T_I ∈ Z^n
for i ← 1 to TREE_DEPTH do
    /* Find the index of the feature evaluated by the
       current node. Then find its value.                 */
    T_F ←Gather(N_F, T_I)                     // T_F ∈ Z^n
    T_V ←Gather(X, T_f)                       // T_V ∈ R^n
    /* Find the threshold, left child and right child     */
    T_T ←Gather(N_T, T_I)                     // T_T ∈ R^n
    T_L ←Gather(N_L, T_I)                     // T_L ∈ Z^n
    T_R ←Gather(N_R, T_I)                     // T_R ∈ Z^n
    /* Perform logical evaluation. If true pick from T_L;
       else from T_R.                                     */
    T_I ←Where(T_V < T_T, T_L, T_R)           // I ∈ Z^n
end
/* Find label for each leaf node                          */
R ←Gather(N_C, T_I)                           // R ∈ Z^n
```

value for each node, respectively. For leaf nodes, we set N_F to 1 and N_T to 0. Finally, N_C captures the class label of each leaf node. For internal nodes this can be any value; we set it to 0.

Given these tensors, Algorithm 2 presents how we perform scoring for a batch of input records X. We use `Gather` and `Where` operations which can be used to perform index-based slicing and conditional value selection. We first initialize an index tensor T_I corresponding to all records in X, which points to the root node. Using T_I, we `Gather` the corresponding feature indices and use them to `Gather` the corresponding feature values from X. Similarly, we also `Gather` left node indices, right node indices, and node thresholds. Using these gathered tensors, we then invoke a `Where` operation which checks for the tree node decisions. Based on the evaluation, for each record the `Where` operator either returns the left child index or right child index. To perform full tree scoring, the above steps have to be repeated until we reach a leaf node for all records in X. We exploit the fact that (1) `TREE_DEPTH` is a known property of the input model at compilation time, and (2) all leaf nodes are at a depth $\leq$ `TREE_DEPTH`, to iterate for that fixed number of iterations to ensure that all records have found their corresponding leaf node. Tensors are created in such a way that if one of the indices reaches a leaf node before running for `TREE_DEPTH` iterations, the same class label will keep getting selected. At compile time, we unroll all iterations and remove the `for` loop to improve efficiency. For ensembles, we create tensors for each tree and batch them together. However, between trees the number of nodes and dimensions may differ, so we use the maximum node count for any tree as the dimension and pad the remaining elements.

Strategy 3: PerfectTreeTraversal (PTT). Similar to the previous one, this strategy also mimics the tree traversal. However, here we assume the tree is a *perfect binary tree*. In a perfect binary tree, all internal nodes have exactly two children and all leaf nodes are at the same depth level. Assume we are given a non-perfect binary tree with a `TREE_DEPTH` of D, and L_k is a leaf node which is at a depth of $D_k < D$. To push L_k to a depth D, we replace L_k with a perfect sub-tree of depth $D - D_k$ and map all the leaf nodes of the sub-tree to C_k: the label of the original leaf node. The decision nodes in the introduced sub-tree are free to perform arbitrary comparisons as the outcome is the same along any path. By pushing all leaf nodes at depth $< D$ to a depth of D, we transform the original tree to a perfect tree with the same functionality.

Table 6: Additional notation used in Strategy 3

Symbol	Description
$I' \in \mathbb{Z}^{2^{D-1}}, L' \in \mathbb{Z}^{2^D}$	Internal and leaf nodes of the perfect tree ordered by level.
$N'_F \in \mathbb{Z}^{\lvert I'\rvert}$	$N'_{F_i} = k \iff I'_i$ evaluates F_k
$N'_T \in \mathbb{R}^{\lvert I'\rvert}$	$N'_{T_i} = \texttt{ThresholdValue}(I'_i)$
$N'_C \in \mathbb{Z}^{\lvert L'\rvert \times \lvert C\rvert}$	$N'_{C_{i,k}} = \begin{cases} 1, N_i \xrightarrow{map\ to} C_k \\ 0, \text{Otherwise} \end{cases}$

Working on perfect trees enables us to get rid of N_L and N_R tensors as we can now calculate them analytically, which also reduces memory lookup overheads during scoring. Thus we create only three tensors to capture the tree structure: N'_F, N'_T, and N'_C (Table 6). They capture the same information as N_F, N_T, N_C but have different dimensions and have a strict condition on the node order. Both N'_F and N'_T have 2^{D-1} elements and the values correspond to internal nodes generated by level order tree traversal. N'_C has 2^D elements with each corresponding to an actual leaf node from left to right order.

Given these tensors, in Algorithm 3 we present how PTT works. From a high-level point of view, it is very similar to the TT strategy with only a few changes. First, the index tensor T_I is initialized to all ones as the root node is always the first node. Second, we get rid of finding the left index and right index of a node and using them in the `Where` operation. Instead, the `Where` operation returns 0 for true case and 1 for

Algorithm 3 PTT Strategy (Notation in Tables 6)

Input : $X \in \mathbb{R}^{n \times |F|}$, Input records
Output : $R \in \{0,1\}^{n \times |C|}$, Predicted class labels

```
/* Initialize all records to point to the root node.      */
T_I ← {1}^n                                          // T_I ∈ Z^n
for i ← 1 to TREE_DEPTH do
    /* Find the index of the feature evaluated by the
       current node. Then find its value.                */
    T_F ←Gather(N_F, T_I)                            // T_F ∈ Z^n
    T_V ←Gather(X, T_f)                              // T_V ∈ R^n
    /* Find the threshold                                */
    T_T ←Gather(N_T, T_I)                            // T_T ∈ R^n
    /* Perform logical evaluation. If true pick left child;
       else right child.                                 */
    T_I ← 2 × T_I + Where(T_V < T_T, 0, 1)           // I ∈ Z^n
end
/* Find label for each leaf node                        */
R ←Gather(N'_C, T_I)                                 // R ∈ Z^n
```

the false case. By adding this to $2 \times T_I$ we get the index of the child for the next iteration. For ensembles, we use the maximum TREE_DEPTH of any tree as D for transforming trees to perfect trees. We create tensors separate for each tree and batch them together for N'_C. But for N'_F and N'_T instead of batching, we interleave them together in some order such that values corresponding to level i for all trees appear before values corresponding to level $i+1$ of any tree.

4.2 Summary of Other Techniques

Next, we discuss the other techniques used across ML operators to efficiently compile them into tensor computations.

Exploiting Automatic Broadcasting. Broadcasting [21] is the process of making two tensors shape compatible for element-wise operations. Two tensors are said to be shape compatible if each dimension pair is the same, or one of them is 1. At execution time, tensor operations implicitly repeat the size 1 dimensions to match the size of the other tensor, without allocating memory. In HB, we heavily use this feature to execute some computation over multiple inputs. For example, consider performing an one-hot encoding operation over column $X_i \in \mathbb{R}^n$ with a vocabulary $V \in \mathbb{Z}^m$. In order to implement this using tensor computations, we Reshape X_i to $[n,1]$ and V to $[1,m]$ and calculate $R = \texttt{Equal}(X, V)$, $R \in \{0,1\}^{n \times m}$. The Reshape operations are for free because they only modify the metadata of the tensor. However, this approach performs redundant comparisons as it checks the feature values from all records against all vocabulary values.

Minimize Operator Invocations. Given two approaches to implement an ML operator, we found that often picking the one which invokes fewer operators outperforms the other—even if it performs extra computations. Consider a featurizer that generates feature interactions. Given an input $X \in \mathbb{R}^{n \times d}$, with $d = |F|$, it generates a transformed output $R \in \mathbb{R}^{n \times \frac{d \cdot (d+1)}{2}}$ with $R_i = [X^2_{i,1}, ..., X^2_{i,d}, X_{i,1}X_{i,2}, ...X_{i,d-1}X_{i,d}]$. One way to implement this operator is to compute each new feature separately by first Gathering the corresponding input feature columns, perform an element-wise Multiplication, and conCatenate all new features. However, this approach requires performing $d^2 + d + 1$ operations and hence is highly inefficient due to high operator scheduling overheads. Alternatively, one could implement the same operator as follows. First, Reshape X into $X' \in \mathbb{R}^{n \times d \times 1}$ and $X'' \in \mathbb{R}^{n \times 1 \times d}$. Then perform a batched GEMM using these inputs, which will create $R' \in \mathbb{R}^{n \times d \times d}$. Finally, Reshape R' to $R'' \in \mathbb{R}^{n \times d^2}$. Notice that each row in R'' has all the values of the corresponding row in R, but in a different order. It also has some redundant values due to commutativity of multiplication (i.e., $x_i x_j = x_j x_i$). Hence, we perform a final Gather to extract the features in the required order, and generate R. Compared to the previous one, this approach increases both the computation and the memory footprint roughly by a factor of two. However, we can implement feature interaction in just two tensor operations.

Avoid Generating Large Intermediate Results. Automatic broadcasting in certain cases can become extremely inefficient due to the materialization of large intermediate tensors. Consider the Euclidean distance matrix calculation, which is popular in many ML operators (e.g., SVMs, KNN). Given two tensors $X \in \mathbb{R}^{n \times d}$ and $Y \in \mathbb{R}^{m \times d}$, the objective is to calculate a tensor $D \in \mathbb{R}^{n \times m}$, where $D_{i,j} = ||X_i - Y_j||_2^2$. Implementing this using broadcasting requires first reshaping X to $X' \in \mathbb{R}^{n \times 1 \times d}$, Y to $Y' \in \mathbb{R}^{1 \times m \times d}$, calculate $(X' - Y') \in \mathbb{R}^{n \times m \times d}$, and perform a final Sum over the last dimension. This approach causes a size blowup by a factor of d in intermediate tensors. Alternatively, a popular trick [37] is to use the quadratic expansion of $D_{i,j} = ||X_i||_2^2 + ||Y_j||_2^2 - 2 \cdot X_i Y_j^T$ and calculate the individual terms separately. This avoids generating intermediate tensors.

Fixed Length Restriction on String Features. Features with strings of arbitrary lengths pose a challenge for HB. Strings are commonly used in categorical features, and operators like one-hot encoding and feature hashing natively support strings. To support string features, HB imposes a fixed length restriction, with the length being determined by the max size of any string in the vocabulary. Vocabularies are generated during training and can be accessed at compile time by HB. Fixed length strings are then encoded into an int8.

5 Optimizations

In this section we discuss the key optimizations performed by the HB's Optimizer: heuristics for picking operator strategies (Section 5.1) and runtime-independent optimizations (Section 5.2). Recall that our approach also leverages runtime-specific optimizations at the Tensor Compiler level. We refer to [8, 41] for runtime-specific optimizations.

5.1 Heuristics-based Strategy Selection

For a given classical ML operator, there can be more than one compilation strategy available. In the previous section we explained three such strategies for tree-based models. In practice, no strategy consistently dominates the others, but each is preferable in different situations based on the input

and model structure. For instance, the GEMM strategy gets significantly inefficient as the size of the decision trees gets bigger because of the large number of redundant computations. This strategy performs $O(2^D)$ (D is the depth of the tree) computations whereas the original algorithmic operator needs to perform only $O(D)$ comparisons. Nevertheless, with small batch sizes or a large number of smaller trees, this strategy can be performance-wise optimal on modern hardware, where `GEMM` operations can run efficiently. With large batch sizes and taller trees, TT techniques typically outperform the GEMM strategy and PTT is slightly faster than vanilla TT due to the reduced number of memory accesses. But if the trees are too deep, we cannot implement PTT because the $O(2^D)$ memory footprint of the associated data structures will be prohibitive. In such cases, we resort to TT. The exact crossover point where GEMM strategy outperforms other strategies is determined by the characteristics of the tree model (e.g., number of trees, maximum depth of the trees), runtime statistics (e.g., batch size), and the underlying hardware (e.g., CPUs, GPUs). For instance, from our experiments (see Figure 8) we found that the GEMM strategy performs better for shallow trees ($D \leq 3$ on CPU, ≤ 10 on GPU) or for scoring with smaller batch sizes. For tall trees, using PTT when $D \leq 10$ give a reasonable trade-off between memory footprint and runtime, which leaves vanilla TreeTraversal the only option for very tall trees ($D > 10$). These heuristics are currently hard-coded.

5.2 Runtime-independent Optimizations

We discuss two novel optimizations, which are unique to HB. HB's approach of separating the prediction pipeline from training pipeline, and representing them in a logical DAG before compilation into tensor computations facilitate the optimization of end-to-end pipelines.

Feature Selection Push-Down. Feature selection is a popular operation that is often used as the *final featurization step* as it reduces over-fitting and improves the accuracy of the ML model [44]. However, during scoring, it can be pushed down in the pipeline to avoid redundant computations such as scaling and one-hot encoding for discarded features or even reading the feature at all. This idea is similar to the concept of projection push-down in relation query processing but through user-defined table functions, which in our case are the ML operators. For operators such as feature scaling, which performs 1-to-1 feature transformations, selection push-down can be easily implemented. However, for operators such as one-hot encoding and polynomial featurization, which perform 1-to-m or m-to-1 feature transformations, the operator will have to absorb the feature selection and stop generating those features. For example, say one-hot encoding is applied on a categorical feature column which has a vocabulary size of 10, but 4 of those features are discarded by the feature selector. In such cases, we can remove such features from the vocabulary. Note that for some "blocking" operators [55], such as normalizers, it is not possible to push-down the feature selection.

Feature Selection Injection. Even if the original pipeline doesn't have a feature selection operator, it is possible to inject one and then push it down. Linear models with L1 regularization (Lasso) is a typical example where feature selection is implicitly performed. The same idea can be extended to tree-based models to prune the features that are not used as decision variables. In both of these examples, the ML model also has to be updated to take into account the pruned features. For linear models we prune the zero weights; for tree models, we update the indices of the decision variables.

6 Experimental Evaluation

In our experimental evaluation we report two micro-benchmark experiments showing how HB performs compared to current state-of-the-art for inference over (1) tree ensembles (Section 6.1.1); (2) other featurization operators and ML models (Section 6.1.2). Then we evaluate the optimizations by showing: (1) the need for heuristics for picking the best tree-model implementation (Section 6.2.1); and (2) the benefits introduced by the runtime-independent optimizations (Section 6.2.2). Finally, we conduct an end-to-end evaluation using pipelines (Section 6.3). We evaluate both CPUs and hardware accelerators (GPUs).

Hardware and Software Setup. For all the experiments (except when stated otherwise) we use an Azure NC6 v2 machine equipped with 112 GB of RAM, an Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an NVIDIA P100 GPU. The machine runs Ubuntu 18.04 with PyTorch 1.3.1, TVM 0.6, scikit-learn 0.21.3, XGBoost 0.9, LightGBM 2.3.1, ONNX runtime 1.0, RAPIDS 0.9, and CUDA 10. We run TVM with `opt_level` 3 when not failing; 0 otherwise.

Experimental Setup. We run all the experiments 5 times and report the truncated mean (by averaging the middle values) of the processor time. In the following, we use ONNX-ML to indicate running an ONNX-ML model (i.e., traditional ML part of the standard) on the ONNX runtime. Additionally, we use **bold numbers** to highlight the best performance for the specific setup (CPU or GPU). *Note that both scikit-learn and ONNX-ML do not natively support hardware acceleration.*

6.1 Micro-benchmarks

6.1.1 Tree Ensembles

Setup. This experiment is run over a set of popular datasets used for benchmarking gradient boosting frameworks [22]. We first do a 80%/20% train/test split over each dataset. Successively, we train a scikit-learn *random forest*, *XGBoost* [40], and *LightGBM* [51] models using the default parameters of the benchmark. Specifically, we set the number of trees to 500 and maximum depth to 8. For XGBoost and LightGBM we use the scikit-learn API. Note that each algorithm generates trees with different structures, and this experiment helps with understanding how HB behaves with various tree types and dataset scales. For example, XGBoost generates balanced

trees, LightGBM mostly generates skinny tall trees, while random forest is a mix between the two. Finally, we score the trained models over the test dataset using different batch sizes. We compare the results against HB with different runtime backends and an ONNX-ML version of the model generated using ONNXMLTools [18]. When evaluating over GPU, we also compared against NVIDIA RAPIDS Forest Inference Library (FIL) [29]. We don't compare against GPU implementations for XGBoost or LightGBM because we consider FIL as state-of-the-art [19]. For the CPU experiments, we use all six cores in the machine, while for request/response experiments we use one core. We set a timeout of 1 hour for each experiment.

Datasets. We use 6 datasets from NVIDIA's gbm-bench [22]. The datasets cover a wide spectrum of use-cases: from regression to multiclass classification, from 285K rows to 100M, and from few 10s of columns to 2K.

List of Experiments. We run the following set of experiments: (1) batch inference, both on CPU and GPU; (2) request/response where one single record is scored at a time; (3) scaling experiments by varying batch sizes, both over CPU and GPU; (4) evaluation on how HB behaves on different GPU generations; (5) dollar cost per prediction; (6) memory consumption; (7) validation of the produced output wrt scikit-learn; and finally (8) time spent on compiling the models.

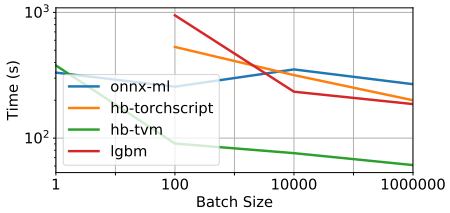
Batch Inference. Table 7 reports the inference time for random forest, XGBoost and LightGBM models run over the 6 datasets. The batch size is set to 10K records. Looking at the CPU numbers from the table, we can see that:

1. Among the baselines, scikit-learn models outperform ONNX-ML implementations by 2 to 3×. This is because ONNX-ML v1.0 is not optimized for batch inference.
2. Looking at the HB's backends, there is not a large difference between PyTorch and TorchScript, and in general these backends perform comparable to ONNX-ML.
3. The TVM backend provides the best performance on 15 experiments out of 18. In the worst case TVM is 20% slower (than scikit-learn); in the best cases it is up to 2× faster compared to the baseline solutions.

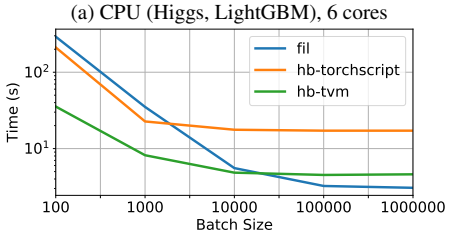
Let us look now at the GPU numbers of Table 7:

1. Baseline RAPIDS does not support random forest nor multiclass classification tasks. For the remaining experiments, GPU acceleration is able to provide speedups of up to 300× compared to CPU baselines.[2]
2. Looking at HB backends, TorchScript is about 2 to 3× slower compared to RAPIDS. TVM is instead the faster solution on 14 experiments out of 18, with a 10% to 20% improvement wrt RAPIDS.

[2]The original FIL blog post [19] claims GPU acceleration to be in the order of 28× for XGBoost, versus close to 300× in our case (Airline). We think that the difference is in the hardware: in fact, they use 5 E5-2698 CPUs for a total of 100 physical cores, while we use a E5-2690 CPU with 6 (virtual) physical cores. Additionally, they use a V100 GPU versus a P100 in our case.



(a) CPU (Higgs, LightGBM), 6 cores



(b) GPU (Airline, LightGBM)

Figure 4: Performance wrt scaling the batch size.

The results are somehow surprising: HB targets the high-level tensor APIs provided by PyTorch and TVM, and still it is able to outperform custom C++ and CUDA implementations.

Request/response. In this scenario, one single record is scored at a time. For this experiment we run inference over the entire test datasets, but with batch size equal to 1. We used the same datasets and setup of Section 6.1.1, except that (1) we removed the Airline dataset since no system was able to complete within the 1 hour timeout; and (2) we only use one single core. The results are depicted in Table 8:

1. Unlike the batch scenario, ONNX-ML is much faster compared to scikit-learn, in some cases even more than 100×. The reason is that ONNX-ML is currently optimized for single record, single core inference, whereas scikit-learn design is more towards batch inference.
2. PyTorch and TorchScript, again, behave very similarly. For random forest they are faster than scikit-learn but up to 5× slower compared to ONNX-ML. For LightGBM and XGBoost they are sometimes on par with scikit-learn, sometime slower.
3. TVM provides the best performance in 11 cases out of 15, with a best case of 3× compared to the baselines.

These results are again surprising, considering that tensor operations should be more optimized for bulk workloads rather than request/response scenarios.

Scaling the Batch Size. We study how the performance of baselines and HB's backends change with the batch size. Figures 4a and 4b depicts the performance variation over CPU and GPU, respectively. We report only a few combinations of dataset / algorithm, but all the other combinations behave similarly. Starting with the CPU experiment, we can see that ONNX-ML has the best runtime for batch size of 1, but then its performance remains flat as we increase the batch size. TorchScript and scikit-learn did not complete within the timeout for batch equal to 1, but, past 100, they both scale linearly as we increase the batch size. TVM is comparable to ONNX-ML for batch of 1; for batches of 100 records it gets about

Table 7: Batch Experiments ($10K$ records at-a-time) for both CPU (6 cores) and GPU. Reported numbers are in seconds.

Algorithm	Dataset	Baselines (CPU)		HB CPU			Baselines (GPU)	HB GPU	
		Sklearn	ONNX-ML	PyTorch	TorchScript	TVM	RAPIDS FIL	TorchScript	TVM
Rand. Forest	Fraud	**2.5**	7.1	8.0	7.8	3.0	not supported	0.044	**0.015**
	Epsilon	9.8	18.7	14.7	13.9	**6.6**	not supported	**0.13**	**0.13**
	Year	1.9	6.6	7.8	7.7	**1.4**	not supported	0.045	**0.026**
	Covtype	**5.9**	18.1	17.22	16.5	6.8	not supported	0.11	**0.047**
	Higgs	**102.4**	257.6	314.4	314.5	118.0	not supported	1.84	**0.55**
	Airline	1320.1	timeout	timeout	timeout	**1216.7**	not supported	18.83	**5.23**
LightGBM	Fraud	3.4	5.9	7.9	7.6	**1.7**	**0.014**	0.044	**0.014**
	Epsilon	10.5	18.9	14.9	14.5	**4.0**	0.15	0.13	**0.12**
	Year	5.0	7.4	7.7	7.6	**1.6**	**0.023**	0.045	0.025
	Covtype	51.06	126.6	79.5	79.5	**27.2**	not supported	0.62	**0.25**
	Higgs	198.2	271.2	304.0	292.2	**69.3**	0.59	1.72	**0.52**
	Airline	1696.0	timeout	timeout	timeout	**702.4**	5.55	17.65	**4.83**
XGBoost	Fraud	1.9	5.5	7.7	7.6	**1.6**	**0.013**	0.44	0.015
	Epsilon	7.6	18.9	14.8	14.8	**4.2**	0.15	0.13	**0.12**
	Year	3.1	8.6	7.6	7.6	**1.6**	**0.022**	0.045	0.026
	Covtype	42.3	121.7	79.2	79.0	**26.4**	not supported	0.62	**0.25**
	Higgs	126.4	309.7	301.0	301.7	**66.0**	0.59	1.73	**0.53**
	Airline	1316.0	timeout	timeout	timeout	**663.3**	5.43	17.16	**4.83**

Table 8: Request/response times in seconds (one record at-a-time).

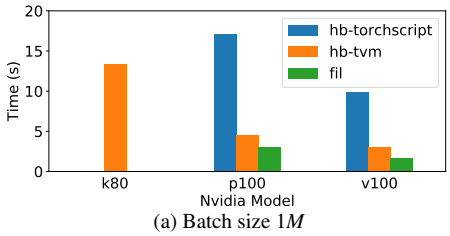
Algorithm	Dataset	Baselines		HB		
		Sklearn	ONNX-ML	PT	TS	TVM
Rand. Forest	Fraud	1688.22	**9.96**	84.95	75.5	11.63
	Epsilon	2945.42	32.58	153.32	134.17	**20.4**
	Year	1152.56	18.99	84.82	74.21	**9.13**
	Covtype	3388.50	35.49	179.4	157.8	**34.1**
	Higgs	timeout	**335.23**	timeout	timeout	450.65
LightGBM	Fraud	354.27	12.05	96.5	84.56	**10.19**
	Epsilon	40.7	29.28	167.43	148.87	**17.3**
	Year	770.11	16.51	84.55	74.05	**9.27**
	Covtype	135.39	209.16	854.07	822.93	**42.86**
	Higgs	timeout	**374.64**	timeout	timeout	391.7
XGBoost	Fraud	79.99	**7.78**	96.84	84.61	10.21
	Epsilon	121.21	27.51	169.03	148.76	**17.4**
	Year	98.67	17.14	85.23	74.62	**9.25**
	Covtype	135.3	197.09	883.64	818.39	**43.65**
	Higgs	timeout	585.89	timeout	timeout	**425.12**

Table 9: Peak memory consumption (in MB) for Fraud.

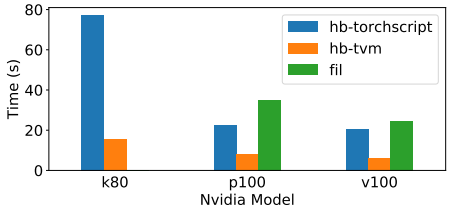
Framework	Random Forest	LightGBM	XGBoost
Sklearn	180	182	392
ONNX-ML	265	258	432
TorchScript	375	370	568
TVM	568	620	811

5× faster, while it scales like TorchScript for batches greater than 100. This is likely due to the fact that TVM applies a set of optimizations (e.g., operator fusion) that introduce a constant-factor speedup compared to TorchScript.

Looking at the GPU numbers (Figure 4b), TorchScript and TVM again follow a similar trend, with TVM being around 3× faster than TorchScript. Both TVM and TorchScript plateau at about a batch size of $10K$. RAPIDS FIL is



(a) Batch size $1M$



(b) Batch size $1K$

Figure 6: Performance across GPUs for Airline, LightGBM

slower than TorchScript for small batch sizes, but it scales better than HB. This is because of its custom CUDA implementation that is able to better use hardware under higher utilization. Interestingly, FIL as well plateaus at around $100K$ records. The custom CUDA implementation introduces a 50% gain over HB with TVM runtime over large batches.

Scaling Hardware. We tested how RAPIDS FIL and HB (TorchScript and TVM) scale as we change the GPU model. For this experiment we tried both with a large batch size ($1M$ records, Figure 6 (a)) to maximize hardware utilization, and a smaller batch size ($1K$, Figure 6 (b)). We ran this on all datasets across random forest, LightGBM, XGBoost with similar results, and present the Airline dataset (the largest) with LightGBM as a representative sample. We tested on three NVIDIA devices: K80 (the oldest, 2014), P100 (2016), and V100 (2017). From the figures, in general we can see that:

Table 10: Conversion times (in seconds) over one core.

Algorithm	Dataset	ONNX-ML	HB		
			PyTorch	TorchScript	TVM
Rand.Forest	Fraud	1.28	0.55	0.58	102.37
	Epsilon	7.53	2.63	2.67	108.64
	Year	7.11	2.77	2.86	69.99
	Covtype	9.87	2.16	2.2	106.8
	Higgs	8.25	2.41	2.44	103.77
	Airline	6.82	2.42	2.53	391.07
LightGBM	Fraud	1.34	0.98	1.06	3.42
	Epsilon	11.71	7.55	7.60	9.95
	Year	9.49	6.11	6.15	8.35
	Covtype	32.46	22.57	23.12	26.36
	Higgs	6.73	25.04	26.3	109
	Airline	11.52	6.38	6.47	8.19
XGBoost	Fraud	0.55	0.65	0.7	86.59
	Epsilon	6.86	25.89	25.94	113.4
	Year	5.66	23.4	23.54	110.24
	Covtype	9.87	2.16	2.20	106.8
	Higgs	6.73	25.04	26.3	109

(1) RAPIDS FIL does not run on the K80 because it is an old generation; (2) with a batch size of 1K we get slower total inference time because we don't utilize the full hardware; (3) TorchScript and TVM runtimes for HB scale similarly on different hardware, although TVM is consistently 4 to 7× faster; (4) FIL scales similarly to HB, although it is 50% faster on large batches, 3× slower for smaller batches; (5) TorchScript is not optimal in memory management because for batches of 1*M* it fails on the K80 with an OOM exception. Finally, we also were able to run HB on the new Graphcore IPU [15] over a single decision tree.

Cost. Figure 7 shows the cost comparison between the Azure VM instance equipped with GPU, and a comparable one without GPU (E8 v3). The plot shows the cost of executing 100k samples with a batch size of 1K for random forest. The cost is calculated based on the hourly rate of each VM divided by the amortized cost of a single prediction. We executed scikit-learn on the CPU and TorchScript and TVM on the GPU for comparison. We found that the CPU cost was significantly higher (between 10×-120×) across all experiments. [3] An interesting result was that the oldest GPU was the most cost effective, with the K80 and TVM having the lowest cost for 13 out of the 18 experiments (including LightGBM and XGBoost, not pictured). This result is explained by the fact that the K80 is readily available at significantly lower cost.

Memory Consumption. We measured the peak memory consumption over the Fraud dataset and for each algorithm. We used the `memory_usage` function in the `memory_profiler` library [2]. The numbers are reported in Table 9, and are the result of the execution over 1 core with a batch size of 1*K*. As we can see, scikit-learn is always the most memory efficient. ONNX-ML consumes from 10% to 50% more memory, while HB with TorchScript runtime consumes from 50% to about 2× more memory than scikit-learn. Conversely, TVM consumes from 2× to 3× more memory wrt scikit-learn. We think that TVM is more memory hungry because it optimizes compute at the cost of memory requirements. Note that the batch size influences the total memory consumption.

[3] Note: airline times out for random forest for CPU with 1*K* batch.

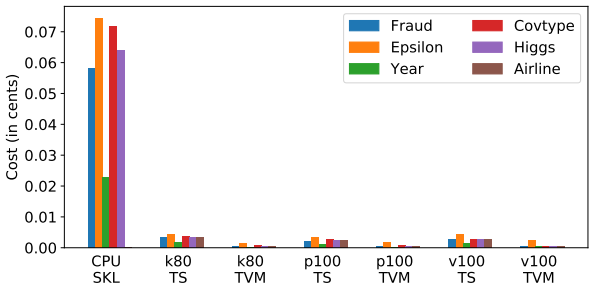


Figure 7: Cost for random forest 100k samples, batch size of 1*K*.

Output Validation. Since we run tree ensemble models as tensor operations, we could introduce rounding errors over floating point operations. Therefore, we need to validate that indeed the outputs produced match. To evaluate this, we used the `numpy testing.assert_allclose` function, and we set the relative and absolute errors to 10^{-5}. We validate both the final scores and the probabilities (when available) for all combinations of datasets and algorithms. Out of the 18 experiments listed in Table 7, 9 of them returned no mismatches for HB, 12 in the ONNX-ML case. Among the mismatches, the worst case for HB is random forest with Covtype where we have 0.8% of records differing from the original scikit-learn output. For the Epsilon dataset, HB with random forest returns a mismatch on 0.1% of records. All the remaining mismatches effect less than 0.1% of records. Note that the differences are small. The biggest mismatch is of 0.086 (absolute difference) for Higgs using LightGBM. For the same experiment ONNX-ML has an absolute difference of 0.115.

Conversion Time. Table 10 shows the time it takes to convert a trained model into a target framework. The numbers are related to the generation of models running on a single core. This cost occurs only once per model and are not part of the inference cost. As we can see, converting a model to ONNX-ML can take up to a few tens of seconds; HB with PyTorch backend is constantly about 2× to 3× faster wrt ONNX-ML in converting random forests models, while it varies for LightGBM and XGBModels. TorchScript models are generated starting from PyTorch models, and in general this further compilation step does not introduce any major overhead. Finally, conversion to TVM is much slower, and it might take more than 3 minutes. This is due to code generation and optimizations introduced in TVM.

As a final note: parallel (i.e., more than 1 core) and GPU execution introduced further conversion time overheads, especially on TVM. For instance, TVM can take up to 40 minutes to convert a random forest model for execution on GPU.

6.1.2 Operators

Setup. This micro-benchmark is a replication of the suite comparing scikit-learn and ONNX-ML operators [17]. We test all scikit-learn operators of the suite that are supported by both ONNX-ML and HB (minus tree ensembles models). The total number of tested operators is 13, and they are a mix of ML models (Logistic Regression, Support Vector Machines, etc.) and featurizers (e.g., Binarizer, Polynomial, etc.). For this micro-benchmark we score 1 million records.

Datasets. We use the Iris datasets [23] with 20 features.

List of Experiments. We run the following experiments: (1) batch inference over 1*M* records, both on CPU and GPU; (2) request/response over 1 record; (3) memory consumption and conversion time. All the output results are correct.

Table 11: Batch experiments for operators on both CPU (1 core) and GPU. Numbers are in milliseconds. (TS is short for TorchScript)

Operator	Baselines (CPU)		HB CPU		HB GPU	
	Sklearn	ONNX-ML	TS	TVM	TS	TVM
Log. Regres.	970	1540	260	**47**	**13**	15
SGDClass.	180	1540	270	**49**	**11**	15
LinearSVC	110	69	260	**51**	**12**	18
NuSVC	3240	4410	**2800**	3000	140	**72**
SVC	1690	2670	**1520**	1560	120	**41**
BernoulliNB	280	1670	290	**65**	**12**	14
MLPClassifier	930	1860	**910**	1430	**17**	31
Dec.TreeClass.	59	1610	560	**35**	**13**	16
Binarizer	98	75	**39**	59	**38**	**38**
MinMaxScaler	92	200	78	**57**	**38**	**38**
Normalizer	94	140	**83**	97	**39**	40
Poly.Features	4030	29160	6380	**3130**	**340**	error
StandardScaler	150	200	77	**58**	**38**	**38**

Batch Inference. The batch numbers are reported in Table 11. On CPU, scikit-learn is faster than ONNX-ML, up to 6× for polynomial featurizer, although in most of the cases the two systems are within a factor of 2. HB with TorchScript backend is competitive with scikit-learn, whereas with TVM backend HB is faster on 8 out of 13 operators, with in general a speedup of about 2× compared to scikit-learn. If now we focus to the GPU numbers, we see that HB with TorchScript backend compares favorably against TVM on 11 operators out of 13. This is in contrast with the tree ensemble micro-benchmark where the TVM backend was faster than the TorchScript one. We suspect that this is because TVM optimizations are less effective on these "simpler" operators. For the same reason, GPU acceleration does not provide the speedup we instead saw for the tree ensemble models. In general, we see around 2× performance improvement over the CPU runtime: only polynomial featurizer runs faster, with almost a 10× improvement. TVM returns a runtime error when generating the polynomial featurizer model on GPU.

Request/response. Table 12 contains the times to score 1 record. The results are similar to the request/response scenario for the tree ensemble micro-benchmark. Namely, ONNX-ML outperform both scikit-learn and HB in 9 out of 13 cases. Note, however, that all frameworks are within a factor of 2. The only outlier is polynomial featurizer which is about 10× faster on HB with TVM backend.

Table 12: Request/Response experiments for operators on CPU (single core). Reported numbers are in milliseconds.

Operator	Baselines		HB	
	Sklearn	ONNX-ML	TS	TVM
LogisticRegression	0.087	**0.076**	0.1	0.1
SGDClassifier	**0.098**	0.1	0.12	0.1
LinearSVC	0.077	**0.05**	0.11	0.1
NuSVC	0.086	**0.072**	4.1	0.14
SVC	0.086	**0.074**	2.3	0.12
BernoulliNB	0.26	**0.1**	0.07	0.11
MLPClassifier	0.15	0.11	**0.1**	0.12
DecisionTreeClassifier	0.087	**0.074**	0.44	0.12
Binarizer	0.064	**0.053**	0.063	0.1
MinMaxScaler	0.066	0.060	**0.058**	0.1
Normalizer	0.11	**0.063**	0.072	0.1
PolynomialFeatures	1.2	1	0.5	**0.1**
StandardScaler	0.069	**0.048**	0.059	0.1

Memory Consumption and Conversion Time. We measured the peak memory consumed and conversion time for each operator on each framework. We used batch inference over 1K records. For memory consumption, the results are in line with what we already saw in Section 6.1.1. Regarding the conversion time, for ONNX-ML and HB with TorchScript, the conversion time is in the order of few milliseconds. The TVM backend is slightly slower but still in the order of few tens of milliseconds (exception for NuSVC and SVC which take up to 3.2 seconds). In comparison with the tree ensembles numbers (Table 10), we confirm that these operators are simpler, even from a compilation perspective.

6.2 Optimizations

6.2.1 Tree Models Implementation

Next we test the different tree-based models implementation to make the case for the heuristics.

Datasets. For this experiment we employ a synthetic dataset randomly generated with 5000 rows and 200 features.

Experiments Setup. We study the behavior of the tree implementations as we change the training algorithm, the batch size, and the tree depth. For each experiment we set the number of trees to 100. We use the TVM runtime backend. Each experiment is run on 1 CPU core.

Results. Figure 8 shows the comparison between the different tree implementations, and the two scikit-learn and ONNX-ML baselines. In the top part of the figure we run all experiments using a batch size of 1; on the bottom part we instead use a batch size of 1*K*. In the column on the left-hand side, we generate trees with a max depth of 3; 7 for the middle column, and 12 for column on the right-hand side. In general, two things are apparent: (1) HB is as fast as or better than the

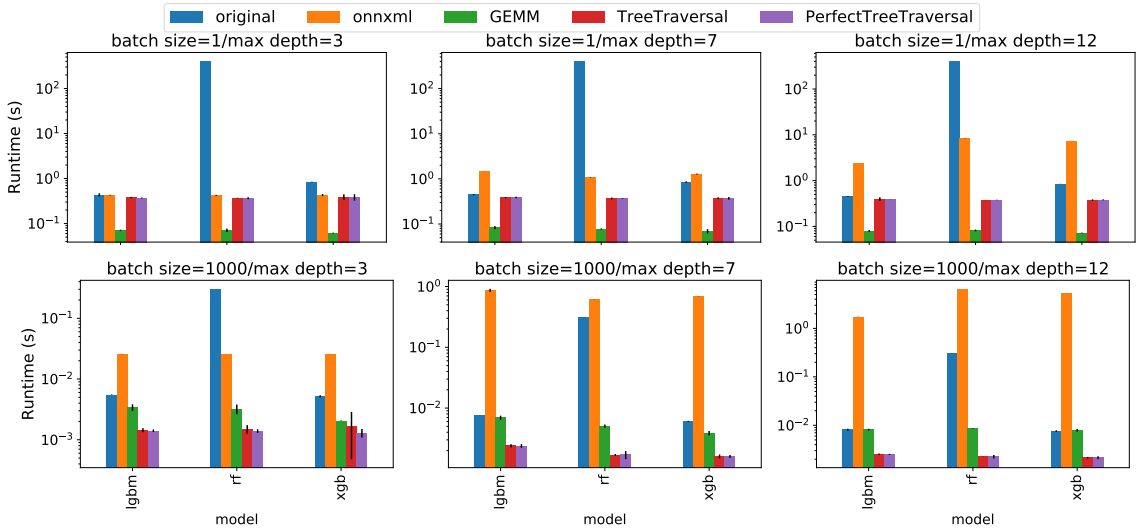


Figure 8: Comparison between the different tree strategies as we vary the batch size and depth.

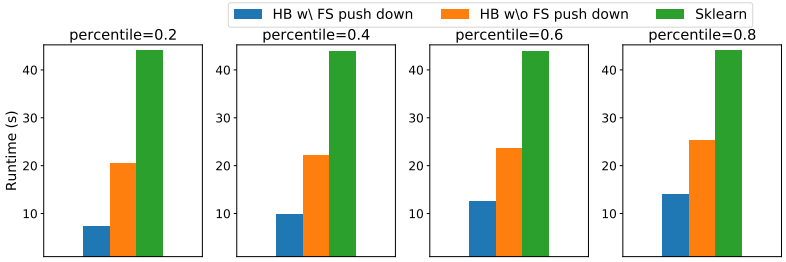


Figure 9: Feature selection push down.

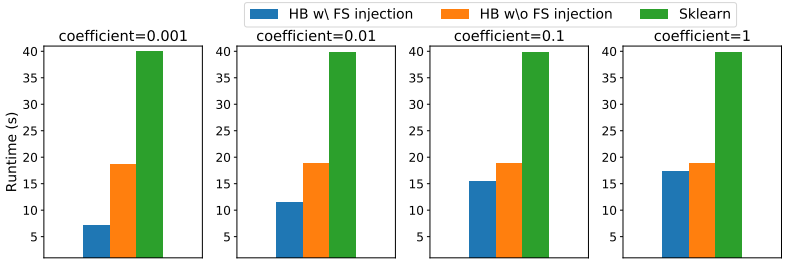
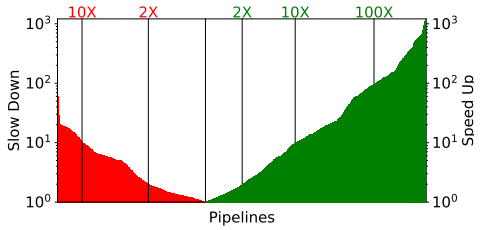
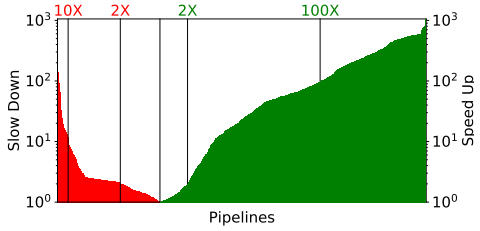


Figure 10: Feature selection injection.



(a) CPU



(b) GPU

Figure 12: Speedup/slowdown of pipelines when using HB wrt baseline Sklearn.

baselines; and (2) no tree implementation is always better than the others. The GEMM implementation outperforms the other two for small batch sizes, whereas TT and PTT are better over larger batch sizes. Between TT and PTT, the latter is usually the best performant (although not by a large margin). PTT however creates balanced trees, and fails for very deep trees.

6.2.2 Runtime-independent Optimizations.

Next we test the optimizations described in Section 5.2.

Dataset. We use the Nomao dataset [24] with 119 features.

Feature Selection Push Down. In this experiment we measure the benefits of the feature selection push down. In Figure 9 we compare HB with and without feature selection push-down, and the baseline implementation of the pipelines in scikit-learn. We use a pipeline which trains a logistic regression model with L2 loss. The featurization part contains one-hot encoding for categorical features, missing value imputation for numerical values, followed by feature scaling, and a final feature selection operator (scikit-learn's `SelectKBest`). We vary the percentile of features that are picked by the feature selection operator. In general, we can see that HB without optimization is about 2× faster than scikit-learn in evaluating the pipelines. For small percentiles, the feature selection push-down optimization delivers a further 3×. As we increase the percentile of features that are selected, the runtime of HB both with and without optimizations increase, although with the optimization HB is still 2× faster than without.

Feature Selection Injection. In this experiment we evaluate whether we can improve the performance of pipelines with sparse models by injecting (and then pushing down) feature selection operators. The pipeline is the same as in the previous case but without the feature selection operator. Instead we train the logistic regression model with L1 regularization. In Figure 10 we vary the L1 regularization coefficient and study how much performance we can gain. Also in this case, with

very sparse models we can see up to 3× improvement wrt HB without optimization. Performance gains dissipate as we decrease the sparsity of the model.

6.3 End-to-end Pipelines

Setup. In this experiment we test HB over end-to-end pipelines. We downloaded the 72 tasks composing the OpenML-CC18 suite [27]. Among all the tasks, we discarded all the "not pure scikit-learn" ML pipelines (e.g., containing also arbitrary Python code). We successively discarded all the pipelines returning a failure during training. 88% of the remaining pipelines are exclusively composed by operators supported by HB, for a total of 2328 ML pipelines. Among these, 11 failed during inference due to runtime errors in HB; we report the summary of executing 2317 pipelines. These pipelines contain an average of 3.3 operators, which is in line with what was observed elsewhere [64].

Datasets. For this experiment we have 72 datasets in total [27]. The datasets are a curated mix specifically designed for ML benchmarking. We did the typical 80%/20% split between training and inference. The smaller dataset has just 100 records, the bigger 19264, while the median value is 462. The minimum number of columns for a dataset is 4, the maximum 3072, with a median of 30.

Results. Figure 12 summarizes the speedup / slowdown introduced by HB when scoring all 2317 pipelines. As we can see, HB is able to accelerate about 60% of the pipelines on CPU (11a). In general, the slowest pipeline gets about 60× slower wrt scikit-learn, the fastest instead gets a 1200× speed up. The slowdowns are due to a couple of factors: (a) the datasets used for these experiments are quite small; (b) some pipelines contain largely sparse operations (i.e., SVM on sparse inputs); (c) several pipelines are small and do not require much computation (e.g., a simple inputer followed by a small decision tree). These three factors are highlighted also by the fact that even if we move computation to the GPU (11b), still 27% of the pipelines have some slowdown. Note however that (1) both sparse and small pipelines can be detected at compile time, and therefore we can return a warning or an error; (2) DNN frameworks are continuously adding new sparse tensor operations (e.g., [34]); and (3) an option could be to add a specific runtime backend for sparse tensor operations (e.g., we have a prototype integration with TACO [52]). In general, DNN frameworks are relatively young, and HB will exploit any future improvement with no additional costs.

With GPU acceleration (Figure 11b), 73% of the pipelines show some speedup. The slowest pipeline gets about 130× slower wrt scikit-learn, the fastest instead gets a speedup of 3 orders of magnitude. Some of the pipelines get worse from CPU to GPU execution. This is due to (1) sparsity; (2) small compute; and (3) data movements between CPU and GPU memory. Indeed we run all pipelines on GPU, even the ones for which in practice would not make much sense (e.g., a decision tree with 3 nodes). We leave as future work an extension to our heuristics for picking the right hardware backend.

7 Related Work

PyTorch [61], TensorFlow [13], MXNet [12], CNTK [10] are DNN frameworks that provide easy-to-use (tensor-based) APIs for authoring DNN models, and heterogeneous hardware support for both training and inference. Beyond these popular frameworks, inference runtimes such as ONNX [5], nGraph [16], TVM [41], and TensorRT [20] provide optimizations and efficient execution targets, specifically for inference. To prove the versatility of our approach, we have tested HB with both PyTorch and TVM. HB uses a two-level, logical-physical optimization approach. First, logical optimizations are applied based on the operators composing the pipeline. Afterwards, physical operator implementations are selected based on model statistics, and physical rewrites, which are externally implemented by the DNN runtime, are executed (e.g., algebraic rewrites, operator fusion). Willump [53] uses a similar two-level optimization strategy, although it targets Weld [60] as its low level runtime and therefore it cannot natively support inference on hardware accelerators. Conversely, HB casts ML pipelines into tensor computations and takes advantage of DNN serving systems to ease the deployment on target environments. Other optimizers for predictive pipelines, such as Pretzel [55], only target logical optimizations. We have integrated HB into Raven [50] as part of our bigger vision for optimizing ML prediction pipelines.

Several works deal with executing trees (ensembles) [29, 59, 67] on hardware accelerators. These systems provide a custom implementation of the PTT strategy specific to the target hardware (e.g., NVIDIA GPUs for RAPIDS FIL [29], FPGAs for [59]), and where computation is parallelized along on the tree-dimension. Alternatively, HB provides three tree inference strategies, including two novel strategies (GEMM and TT), and picks the best alternative based on the efficiency and redundancy trade-off.

8 Conclusions

In this paper, we explore the idea of using DNN frameworks as generic compilers and optimizers for heterogeneous hardware. Our use-case is "traditional" ML inference. We ported 40+ data featurizers and traditional ML models into tensor operations and tested their performance over two DNN frameworks (PyTorch and TVM) and over different hardware (CPUs and GPUs). The results are compelling: even though we target high-level tensor operations, we are able to outperform custom C++ and CUDA implementations. To our knowledge, HUMMINGBIRD is the first system able to run traditional ML inference on heterogeneous hardware.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd, Chen Wenguang, for their feedback and suggestions to improve the paper. We would also like to thank Nellie Gustafsson, Gopal Vashishtha, Emma Ning, and Faith Xu for their support.

References

[1] Cerebras Chip. https://www.wired.com/story/power-ai-startup-built-really-big-chip/.

[2] Memory profiler for Python.

[3] Nvidia RAPIDS. https://developer.nvidia.com/rapids.

[4] ONNX ML. https://github.com/onnx/onnx/blob/master/docs/Operators-ml.md.

[5] ONNX Runtime. https://github.com/microsoft/onnxruntime.

[6] ONNX Supported Frameworks and Backends. https://onnx.ai/supported-tools.html.

[7] Pandas. https://pandas.pydata.org/.

[8] TorchScript Documentation. https://pytorch.org/docs/stable/jit.html.

[9] H2O Algorithms Roadmap. https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/flow/images/H2O-Algorithms-Road-Map.pdf, 2015.

[10] CNTK. https://docs.microsoft.com/en-us/cognitive-toolkit/, 2018.

[11] Matplotlib. https://matplotlib.org/, 2018.

[12] MXNet. https://mxnet.apache.org/, 2018.

[13] TensorFlow. https://www.tensorflow.org, 2018.

[14] Esg technical validation: Dell emc ready solutions for ai: Deep learning with intel. https://www.esg-global.com/validation/esg-technical-validation-dell-emc-ready-solutions-for-ai-deep-learning-with-intel, 2019.

[15] Graphcore IPU. https://www.graphcore.ai/, 2019.

[16] nGraph. https://www.ngraph.ai/, 2019.

[17] ONNX-ML vs Sklearn Benchmark. https://github.com/xadupre/scikit-learn_benchmarks, 2019.

[18] ONNXMLTools. https://github.com/onnx/onnxmltools, 2019.

[19] RAPIDS Forest Inference Library. https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35, 2019.

[20] Tensor-RT. https://developer.nvidia.com/tensorrt, 2019.

[21] Broadcasting Semantic. https://www.tensorflow.org/xla/broadcasting, 2020.

[22] Gradient Boosting Algorithm Benchmark. https://github.com/NVIDIA/gbm-bench, 2020.

[23] Iris dataset. https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html, 2020.

[24] nomao dataset. https://www.openml.org/d/1486, 2020.

[25] ONNX. https://github.com/onnx/onnx/blob/master/docs/Operators.md, 2020.

[26] ONNX Portable format. https://www.infoworld.com/article/3223401/onnx-makes-machine-learning-models-portable-shareable.html, 2020.

[27] OpenML-CC18 Benchmark. https://www.openml.org/s/99, 2020.

[28] Pytorch Ecosystem. https://pytorch.org/ecosystem/, 2020.

[29] RAPIDS cuML. https://github.com/rapidsai/cuml, 2020.

[30] Sambanova: Massive Models for Everyone. https://sambanova.ai/, 2020.

[31] skl2onnx Converter. https://github.com/onnx/sklearn-onnx/, 2020.

[32] Tensorflow JS. https://www.tensorflow.org/js, 2020.

[33] Tensorflow XLA. https://www.tensorflow.org/xla, 2020.

[34] The Status of Sparse Operations in Pytorch. https://github.com/pytorch/pytorch/issues/9674, 2020.

[35] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Gowdal, Matteo Interlandi, Alekh Jindal, Kostantinos Karanasos, Subru Krishnan, Brian Kroth, Jyoti Leeka, Kwanghyun Park, Hiren Patel, Olga Poppe, Fotis Psallidas, Raghu Ramakrishnan, Abhishek Roy, Karla Saur, Rathijit Sen, Markus Weimer, Travis Wright, and Yiwen Zhu. Cloudy with high chance of DBMS: A 10-year prediction for Enterprise-Grade ML. *arXiv e-prints*, page arXiv:1909.00084, Aug 2019.

[36] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. Machine learning at Microsoft with ML.NET. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '19, page 2448–2458, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Samuel Albanie. Euclidean distance matrix trick. 2019.

[38] Amazon. The total cost of ownership (tco) of amazon sagemaker. https://pages.awscloud.com/rs/112-TZM-766/images/Amazon_SageMaker_TCO_uf.pdf, 2020.

[39] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1387–1395, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[41] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 579–594, Berkeley, CA, USA, 2018. USENIX Association.

[42] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: ML inference pipeline composition framework. *CoRR*, abs/1812.01776, 2018.

[43] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.

[44] Manoranjan Dash and Huan Liu. Feature selection for classification. *Intelligent data analysis*, 1(3):131–156, 1997.

[45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, page arXiv:1810.04805, Oct 2018.

[46] FirmAI. Machine Learning and Data Science Applications in Industry. https://github.com/firmai/industry-machine-learning.

[47] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[48] Intel. Machine learning fpga. https://www.intel.com/content/www/us/en/products/docs/storage/programmable/applications/machine-learning.html, 2020.

[49] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[50] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. Extending relational query processing with ML inference. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[51] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[52] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[53] Peter Kraft, Daniel Kang, Deepak Narayanan, Shoumik Palkar, Peter Bailis, and Matei Zaharia. Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference. *arXiv e-prints*, page arXiv:1906.01974, Jun 2019.

[54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[55] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.

[56] Ping Li. Robust logitboost and adaptive base class (abc) logitboost. In *n Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI'10)*.

[57] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 2020.

[58] Faith Xu Matteo Interlandi, Karla Saur. Accelerate traditional machine learning models on GPU with ONNX Runtime. https://cloudblogs.microsoft.com/opensource/2020/09/29/accelerate-machine-learning-models-gpu-onnx-runtime-hummingbird/, 2020.

[59] M. Owaida, H. Zhang, C. Zhang, and G. Alonso. Scalable inference of decision tree ensembles: Flexible design for cpu-fpga platforms. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2017.

[60] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018.

[61] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[62] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[63] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data lifecycle challenges in production machine learning: A survey. *SIGMOD Record*, 47(2):17–28, 2018.

[64] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, and Markus Weimer. Data science through the looking glass and what we found there, 2019.

[65] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. Froid: Optimization of imperative programs in a relational database. *Proc. VLDB Endow.*, 11(4):432–444, December 2017.

[66] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[67] Toby Sharp. Implementing decision trees and forests on a gpu. In David Forsyth, Philip Torr, and Andrew Zisserman, editors, *Computer Vision – ECCV 2008*, pages 595–608, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[68] Credit Suisse. The apps revolution manifesto—volume 1: The technologies. https://aka.ms/enterprise-application-lifespan, 2012.

[69] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

A Artifact Appendix

A.1 Abstract

Hummingbird compiles trained traditional ML models into tensor computation for faster inference. Hummingbird allows users to score models both on CPU and hardware accelerators.

A.2 Artifact check-list

- **Program:** PyTorch, ONNX Runtime, TVM.
- **Data set:** Fraud, Epsilon, Year, Covtype, Higgs, Airline, Iris, Nomao, OpenMLCC-18.
- **Run-time environment:** Ubuntu 18.04.
- **Hardware:** Azure NC6 v2 machine.
- **Experiments:** tree-models (Random Forest, XGBoost, LightGBM), operators (LogisticRegression, SGDClassifier, LinearSVC, NuSVC, SVC, BernoulliNB, MLPClassifier, DecisionTreeClassifier, Binarizer, MinMaxScaler, Normalizer, PolynomialFeatures, StandardScaler), end-to-end pipelines.
- **Public link:** https://github.com/microsoft/hummingbird.
- **Code licenses:** MIT.

A.3 Description

A.3.1 How to access

Hummingbird is open source and can be accessed directly from https://github.com/microsoft/hummingbird. Otherwise, Hummingbird can also be downloaded from pip with `pip install hummingbird-ml`.

A.3.2 Hardware dependencies

No specific hardware dependencies. The artifact has been evaluated on different NVIDIA GPU generations (K80, P100, V100) but it should work on any hardware supported by the target DNN runtime.

A.3.3 Software dependencies

Hummingbird requires Python >= 3.5, numpy>=1.15, onnxconverter-common>=1.6.0, scikit-learn>=0.21.3, torch>=1.3.1. Additional dependencies for reproducing the results are onnxruntime >= 1.0, onnxmltools>=1.6.0, xgboost>=0.90 and lightgbm>=2.2, psutil, memory-profiler.

A.3.4 Data sets

For the experiments on tree algorithms we used Fraud [4], Epsilon [5], Year [6], Covtype [7], Higgs [8], and Airline [9]. For the experiments on operators we instead used Iris [10]. Finally, for the pipeline experiments we used OpenML-CC18 [27]. The experiment scripts automate the download and preparation of all the datasets.

A.4 Installation

Hummingbird can be installed from pip with `pip install hummingbird-ml` or by cloning the code available on GitHub and by calling `python setup.py install` from the main directory. Hummingbird will automatically detect the available backends at runtime. We refer to https://github.com/microsoft/hummingbird/blob/master/TROUBLESHOOTING.md for problems related to installations.

A.5 Experiment workflow

The scripts for the experiments are divided in three main folders: *trees*, *operators* and *pipelines*. Each folder contains a README.md file containing the specific instructions for that particular set of experiments.

Trees: This directory contains the script to generate the result of Section 6.1.1. We suggest to start with running `python run.py -dataset fraud,year,covtype,epsilon` (skipping higgs/airline) because the complete script (which can be run with just `python run.py`) over all backends and datasets takes more than one day to complete. After the script is run for the first time, the datasets and trained models are cached (in datasets and models folders, respectively), so that following executions will be faster. Serveral other arguments can be changed in the script (e.g., batch size, number of trees, etc.).

The output of the above commands is a JSON file reporting the training time and accuracy (if the model is not cached), and prediction (process) time in seconds, as well the peak memory used. The baseline is then compared against Hummingbird with PyTorch (hb-pytorch), TorchScript (hb-torchscript) and TVM (hb-tvm) backends. The entry `is_same_output` specifies whether the results of the translated models match those of the baseline (up to a tolerance of 10^-6). If the result is `false`, the script can be re-run with the `-validate` flag on to check the percentage of wrong results. The `-gpu` flag can be used to run the experiments on GPU.

Operators: This directory contains the scripts to reproduce the experiments of Section 6.1.2. The scripts are configured to run scikit-learn and compare it against ONNX-ML, TorchScript and TVM (the last 2 using Hummingbird), for the Iris dataset over 1 core, and with batch of 1M. `python run.py` runs the benchmark for CPU, `python run.py -gpu` runs the benchmark for GPU.

Pipelines: This directory contains the script to reproduce the experiments of Section 6.3. There are two main scripts to run for this experiment:

- `openml_pipelines.py` is used to download and train all the scikit-learn pipelines of the openML-CC18 benchmark.
- `run.py` is used to run evaluate the performance of scikit-learn and Hummingbird over the trained pipelines.

[4] https://www.kaggle.com/mlg-ulb/creditcardfraud
[5] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html
[6] https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd
[7] https://archive.ics.uci.edu/ml/datasets/covertype
[8] https://archive.ics.uci.edu/ml/datasets/HIGGS
[9] http://kt.ijs.si/elena_ikonomovska/data.html
[10] https://archive.ics.uci.edu/ml/datasets/iris

This experiment is composed of two steps. The first step in this experiment is the generation of the prediction pipelines. This can achieved by running `python openml_pipelines.py | tee openml-cc18.log` This script takes several hours to run. While executing, this script will log the number of successfully trained pipelines, as well as additional statistics. Once completed, the openml-cc18.log file contains the statistics. Per task statistics are logged into the relative folder.

Once the first step is completed, in the second step we evaluate the scoring time of the generated pipelines, and compare the speed-ups introduced by Hummingbird against scikit-learn. This experiment can be executed both on CPU and GPU, and in both cases it takes about an hour. `python run.py` runs inference over all the generated pipelines, while `python run.py -gpu` can be used for GPU execution.

A.6 Evaluation and expected result

In May we open sourced Hummingbird (blog post: https://azuredata.microsoft.com/articles/ebd95ec0-1eae-44a3-90f5-c11f5c916d15). Since then we have been pushing our internal code into the open source repository, but the 2 versions do not match yet. Specifically:

- TVM integration is not complete. In our internal version we re-implemented all the operators directly in TVM's Relay but this is not a good strategy in the long term. In the open source version, we directly export Relay graphs from PyTorch models. However the exporter does not cover PyTorch 100% yet. We are however working with the TVM community for bringing full support of TVM in Hummingbird (we suggest to check the related issue #232 on Hummingbird's GitHub if interested). In practice, this means that: (1) not all operators are currently exportable into TVM; and (2) the performance we reported in the paper for TVM can be a bit different.
- The optimizer is not yet open sourced. This means that Figures 9 and 10 are not reproducible as of now. We hope to be able to bring the optimizer open source in the coming months.

Besides the above two limitations, the scripts allow the reproduction of the following main results of the paper:

- `trees` allows the reproduction of the results of Tables 7, 9 and 10. Please check the above description for specifics.
- `operators` allows the reproduction of the results of Table 11 (however not all operators will run on the TVM backend). Again, please check the related description for specifics.
- `pipelines` allows the reproduction of the results of Figure 12. Also in this case we don't cover yet 100% of the operators, but we are close.

Keep in mind that running all the experiments for completely reproducing the results will take several days.

A.7 Experiment customization

The above mentioned scripts can be customized by running them with different input arguments. For instance, Table 8 in the paper can be reproduced by setting the batch size to 1 (using the `-batch_size` argument.) in the `run.py` script.

A.8 Notes

The numbers in the paper were run on the reported VM, however:

- As this is an Azure VM, the underlying machine can receive upgrades necessitating the reinstallation of the NVidia drivers.
- The original experiments were run inside the context of an Nvidia-docker container. This setup should not have a large impact on results

Additionally, a few operators are not yet available in the open source version of Hummingbird, therefore the final coverage reported in the log file for the pipelines will be different than the one reported in the paper. To check the expected coverage once all the operators are open source, the script allows to add new operators. The same consideration holds for the operators experiment.

As a final note: to allow third-party reproducibility, we are open sourcing all the scripts used for the experiments.

A.9 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts



Retiarii: A Deep Learning Exploratory-Training Framework

Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, Lidong Zhou

Microsoft Research

Abstract

Traditional deep learning frameworks such as TensorFlow and PyTorch support training on a single deep neural network (DNN) model, which involves computing the weights iteratively for the DNN model. Designing a DNN model for a task remains an experimental science and is typically a practice of deep learning model exploration, dovetailed with training and validation, aiming to find the best model among a set that yields the best result. Retrofitting such exploratory-training into the training process of a single DNN model, as supported by current deep learning frameworks, is unintuitive, cumbersome, and inefficient, because of the fundamental mismatch between exploring a set of models and training a single one.

Retiarii is the first framework to support deep learning exploratory-training. In particular, Retiarii (i) provides a new programming interface to specify a DNN model space for exploration, as well as an interface to describe the exploration strategy that decides which order to instantiate and train models in, how to prioritize model training, and when to terminate training of certain models; (ii) offers a Just-In-Time (JIT) engine that instantiates models, manages the training of the instantiated models, gathers the information for the exploration strategy to consume, and executes the decisions accordingly; (iii) identifies the correlations between the instantiated models and develops a set of cross-model optimizations to improve the overall exploratory-training process. Retiarii does so by introducing a key abstraction, *Mutator*, that connects the specifications of DNN model spaces and exploration strategies, while exposing the correlations between models for optimization. As a result, Retiarii's clean separation of DNN model space specification, exploration strategy, and cross-model optimizations, connected through the single mutator abstraction, leads to ease of programming, reuse of components, and vastly improved (up to 8.58x) overall exploratory-training efficiency.

1 Introduction

Deep neural networks (DNNs) have been successfully applied to a variety of perception-based tasks such as vision and speech. For each such task, a DNN model architecture, depicted as a graph of operators as vertices, connected with weighted edges, is designed. The model is then trained to populate the weights, before it can be used to perform the task. Deep learning frameworks, such as TensorFlow [11] and PyTorch [48], have been designed to describe an individual DNN model and train the model as a (training) job to run on target hardware, such as GPUs. Training a deep learning model is often resource intensive and costly.

Devising a model for a particular task often involves an iterative exploration process, where a developer would often start with a model architecture that captures the main intuitions and tweak it repeatedly until a model with satisfactory results is identified in a continuous training and validation process. Alternatively, a model architecture could also evolve from simple models following a simple set of evolution rules.

There are clear gaps between the needs to support this *exploratory-training* process and the existing deep learning frameworks. First, this exploratory-training process works on a series of deep learning models, rather than a single one, as supported by the existing deep learning frameworks. A developer either has to specify each model individually in a manual, tedious, and repetitive process, or encodes this series of models as one "jumbo" model [13, 27, 50, 65] using advanced features such as dynamic graph and control flow. Such a "jumbo" model pollutes the original model architecture and makes it significantly harder to understand as changes are scattered across the model description with complex dynamic, control-flow structures. It is also more difficult to optimize due to the use of those dynamic, control-flow structures.

Second, deep learning frameworks manage individual training jobs and cannot capture or leverage the correlation among the set of training jobs in the same exploratory-training process. A developer is again forced to code certain exploration strategies in a "jumbo" model, together with ad hoc runtime mechanisms to manipulate the priorities of jobs or stop not-so-promising jobs early. Such implementations of exploration strategies are hardly reusable as they are deeply coupled with and embedded in a particular exploratory-training process. And there is no easy way to expose the correlations among those models, which tend to share many common structures, for cross-model optimizations. Training a set of models often incurs significant cost; any efficiency gains through optimizations would often allow an exploratory-training process to find a better model under the same budget.

We therefore propose Retiarii, the first deep learning framework specifically designed to support exploratory-training. To address the gaps we have previously identified in the existing deep learning frameworks, we address three core problems of exploratory-training: (i) specifying a DNN model space to explore, (ii) defining and realizing exploration strategies

to decide when to instantiate a model in the space, which ones to instantiate, how to prioritize the training of the instantiated models, and when to terminate the jobs for training those models, and (iii) exposing the correlations among the instantiated models and optimizing training across models by leveraging the correlation information.

Retiarii embraces a new *Mutator* abstraction as the basis for specifying a DNN model space and for defining an exploration strategy. Observing that the exploratory-training process tends to introduce relatively minor modifications to existing models or to compose simple models together following a set of evolution rules, Retiarii allows developers to specify each such modification or evolution as a *mutator* on a model graph. A DNN model space for an exploratory-training process can be defined as a set of base models (each specified as in the original deep learning frameworks, with no "pollution") and a set of mutators. The DNN mode space is then the base models, plus any subsequent models produced by applying mutators to the current models, and so on. An exploration strategy can then be partly defined to govern when to generate new models by applying mutators, as well as which current models and mutators to choose.

Retiarii further designs a Just-In-Time (JIT) engine for the exploratory-training process, which essentially manages the logical collection of all models and their corresponding training jobs. The engine instantiates new models dynamically, exposes the correlations of the instantiated models for cross-model optimizations, schedules the optimized jobs for execution, and manages the execution of the scheduled jobs, governed by the specified exploration strategy.

Retiarii advocates a clean separation of concerns and strives for simplicity and modularity. The mutator abstraction focuses on the changes to an existing model and exposes the differences (and similarities) of models for cross-model optimizations. Each mutator is fine-grained, to capture a logical unit of modification, and intended to be composable and reusable. The cross-model optimizations are also designed and implemented as general capabilities, enabled by the mutator abstraction, in Retiarii's JIT engine. Exploration strategies are decoupled from the specification of the model spaces (through base models and mutators) to maximize reusability, even though some exploration strategies might unavoidably have dependencies on certain types of model spaces.

We have fully implemented and open sourced Retiarii [1]. So far, Retiarii implements 6 mutators to define 18 different model spaces, 11 different exploration strategies, and 3 cross-model optimizations. These combinations have already covered 27 NAS algorithms from the research community, and benefit from vastly improved performance with cross-model optimizations. Our evaluation shows that (1) Retiarii reduces the exploration time of popular Neural Architecture Search (NAS) algorithms by up to 2.57×, and (2) Retiarii improves the scalability of NAS algorithms using weight sharing with a speed-up of 8.58×.

[1]Source code available at `https://github.com/microsoft/nni/tree/retiarii_artifact`

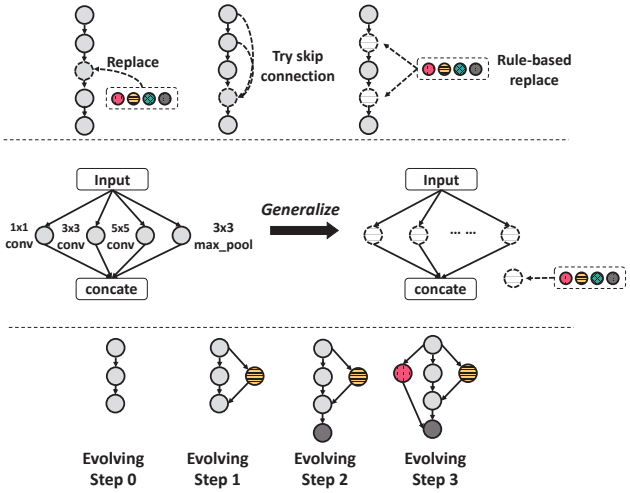


Figure 1: Three typical types of model space explorations.

2 Background and Motivation

The many ways of creating candidate model variations. Developing a model typically involves creating interesting candidate model variations following some design intuitions; for example, by 1) tweaking a substructure (*e.g.,* a layer or a cell) of a base model, 2) coming up with generalized cell structure, 3) or evolving network structure gradually, as shown in Figure 1.

The top set of examples in Figure 1 shows different ways of modifying a base model. One could replace an operator at a layer with some candidate operators (*e.g., normal conv*, *depthwise conv*), or changing a layer's input (*e.g.,* adding some skip connections). The modification can also be applied to a *cell* containing several interconnected layers, but treated as a one logical layer. More generally, a matching rule can be defined to apply modifications on the entire model (*e.g.,* adding *BatchNorm* after convolution layers or replacing all *ShuffleNet* cells [42] with *Inverted Residual* cells [54]).

The middle example in Figure 1 shows how one could generalize a cell structure in order to find a better one. For example, an Inception cell [57] can be generalized to explore a space with different numbers of paths and a different operator on each path. Similarly, an LSTM structure can be generalized to an RNN cell [69]. A generalized structure usually contains a large number of different structures.

The bottom example in Figure 1 shows how the final network gradually evolves from a simple network following some rules. The rules could be adding a layer/edge or changing a layer's operator in each evolution step [23].

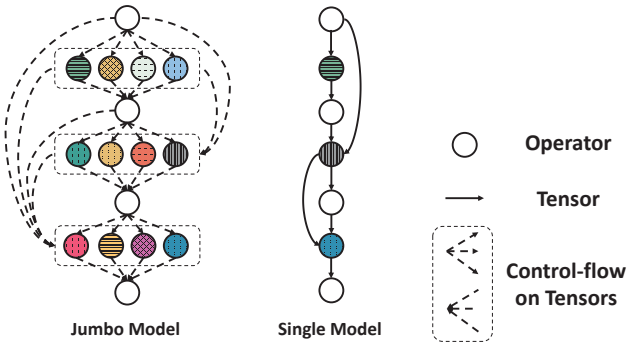


Figure 2: The jumbo model compared to a single model in the model space. Colored circles are different operators.

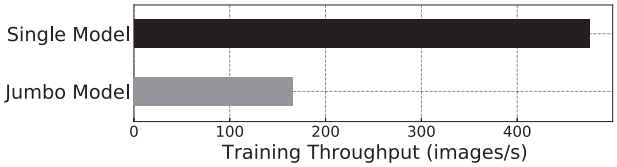


Figure 3: The throughput of ResNet50 built as a single model and a jumbo model. The space contains 4 choices of convolution operator at each layer. Both computation graphs are optimized by TensorFlow XLA [38].

The pain of specifying and exploring a model space. Exploring a model space, as exemplified in Figure 1, is not directly supported by the existing deep learning frameworks, such as TensorFlow and PyTorch. A model developer often has to program and train each model manually, or to code up all the variations of models in a model space as a single *jumbo model* in TensorFlow/PyTorch through complex control-flow, such as using specified values on the condition of control-flows to route to each model [27, 50, 62, 70]. Figure 2 shows a simple example, a layer has four candidate operators (*e.g., normal conv*, *depthwise conv*, *avgpool*, and *maxpool*), there should be a control-flow to pick one during model construction. If a layer's input is the output of one of the previous layers (*e.g.,* skip connection), there should be a dynamic control-flow to route to the right path during model *forward* (*i.e.,* forward pass of data flow graph). Some popular model spaces [50, 62] change operators and inputs on as many as tens of layers, leading to excessive complexity, making the code hard to understand, and going beyond the limited capabilities of current frameworks to handle control-flow. The control-flow in jumbo models also make them hard to apply compiler optimization techniques, such as operator fusion [15, 38] and memory planning [16]. Figure 3 shows the performance gaps in terms of throughout for ResNet50, as a single model vs. as one encoded as part of a jumbo model.

Automatic model exploration. A DNN model space can be explored automatically with an *exploration strategy*. The action scope of exploration strategy spans from model generation to model execution.

When exploring a huge model space, it is usually impossible and unnecessary to train all the models in the space. An exploration strategy is responsible for deciding which models to instantiate and train, in what priority, and when to terminate. A typical strategy on which models to instantiate could be *brute force* (*e.g.,* random search [56] and grid search [60]), *heuristic-based* (*e.g.,* evolution [23, 30] and annealing algorithms [36]), or more advanced *model-based* (*e.g.,* Bayesian models [33, 67] and reinforcement learning [59, 69, 70]).

An exploration strategy further manages the executions of training instantiated model; for example, to stop the execution of a bad-performing model early based on a performance predictor [20], or to dynamically adjust the computation resource provided to each model depending on the model's performance [64], or to run several mini-batches only and share the weights of overlapped layers among the models to reduce each model's execution time significantly [27, 50].

The pain of implementing exploration strategies. An exploration strategy naturally manages a set of models. Implementing such a strategy with the existing deep learning frameworks is unintuitive and cumbersome, as those frameworks are designed for training individual models and have no support for an exploration strategy.

Because an exploration strategy intensively involves instantiating models from a model space, the implementation often tightly couples an exploration strategy with a specific model space, further increasing the complexity of already complicated jumbo models. For example, an RNN-based RL algorithm (a popular exploration strategy) uses each of its time steps to control the condition value of each control-flow in the jumbo model [50]. Further incorporating the logic of controlling model training makes the jumbo model unmanageable. As a result, though most exploration strategies are logically applicable to different model spaces, the implementations embedded in the jumbo models are hardly reusable by other model spaces.

Encoding an exploration strategy in a jumbo model also makes it hard to expose cross-model optimization opportunities as an exploratory-training usually produces many models. The models explored tend to have strong correlations (*e.g.,* common computation logic) among them, as the variations produced tend to touch only a certain part of the model, while keeping the rest unchanged. The training of those models also share the same dataset and data preprocessing logic. To adapt a model to different tasks, the large backbone network (*e.g.,* BERT) is often fixed: the exploration tends to focus on varying the structure of several added layers. Significant opportunities, therefore, exist in leveraging common computation across model training to speed up an exploratory-training process as a whole. When encoding an exploration strategy in

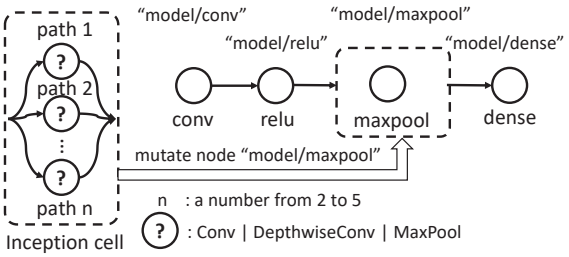


Figure 4: Base model and mutation: An example.

```
create_node(name:str,op:Op,params:dict={})
delete_node(node:Node)
connect(src:NodeOutput,dst:NodeInput)
del_connect(src:NodeOutput,dst:NodeInput)
update_node(node:Node,op:Op=None,params:dict={},
            inputs:list=None)
choose(candidates:list,n_chosen:int=1,
       type:str="choice",ctx:dict=None)
```

Figure 5: Mutation primitives and the `choose` primitive.

a jumbo model, it also becomes challenging to scale the training of this jumbo job to multiple GPUs and servers [27, 50]; in contrast, scaling to multiple GPUs is straightforward for a set of individual jobs, each training a single model.

Insufficiency of existing AutoML systems. Previous AutoML systems (*e.g.,* Google Vizier [24], Auto-WEKA [61], Auto-Sklearn [22]) abstract the AutoML problem as hyper-parameter tuning. Although a certain NAS problem can be modeled as the tuning of specific hyper-parameters, it often involves the definition of an ad-hoc set of hyper-parameters, making it cumbersome to express different model spaces in a general way. It is especially painful to hyper-parameterize evolutionary NAS [13, 23, 51] where neural architectures can randomly evolve. Moreover, the expressed architectures are hardly understood by compilers, making optimizations almost impossible. Some recently emerged AutoML systems (*e.g.,* AutoKeras [32]) provide more support to NAS. They can automatically search neural architectures but with specifically implemented model spaces and exploration strategies, where system optimizations are hardly applicable.

Retiarii is designed to address the abovementioned pains. It provides great expressiveness to support various model spaces and strategies in a systematic and programmable way. It clearly decouples model space from exploration strategy and enables system optimizations to speed up exploration process.

3 Mutator as the Core Abstraction

Exploratory-training is all about exploring a model space. Mutator is the core abstraction that connects the specification and exploration of a model space, while exposing the correlations between models for further optimizations.

Base models. Retiarii follows the standard practice of characterizing a DNN model as a data-flow graph (DFG), where each node represents an operator (or a subgraph) with one or multiple input and output tensors and an edge connects an output tensor of a node to an input tensor of another node.

Retiarii introduces the notion of *base models* as the starting points of an exploratory-training and preserves the way a single DNN model is specified for base models. In fact, Retiarii can simply import base models defined in an existing deep learning framework such as TensorFlow. Figure 4 illustrates an example base model with a chain of 4 operators (a convolutional neural network).

Mutator. Exploratory-training is typically a process of applying modifications (*e.g.,* as depicted in Figure 1) to existing models, starting from base models. Rather than encoding modifications in a complex jumbo model, Retiarii cleanly separates modifications from the original target models and encode each as a Mutator, an abstraction designed to be expressive, modular, reusable, and composable. The model space to be explored by an exploratory-training process is then the base models, plus all the resulting models from applying mutators to the base models and to any subsequently generated models.

Graph matching and manipulation in Mutator. Each mutator specifies matching criteria to identify subgraphs of a target model's DFG to operate on, followed by a series of graph construction operations to modify the matched subgraphs to create a new model. The mutator abstraction can also use the `choose` primitive to describe different options to choose from in a mutator, so that the mutator can produce a number of models without duplicating the mutator code to create a new mutator for each option.

Retiarii's current graph matching is based on node type or node name, which is simple, but sufficient for all the use cases we have implemented. But it can be extended easily to more expressive graph matching if necessary.

Retiarii introduces general mutation primitives like `create_node` for a mutator to manipulate the node and edge in a model. The primitives are summarized in Figure 5. Note that a node in Retiarii can also represent a subgraph. Thus the primitives can also be applied to a subgraph (*e.g.,* a layer or a cell) of a model.

For each model instantiation, Retiarii records all the mutation primitives called in a mutator. Hence Retiarii can easily identify model correlations across instantiated models. For example, between two instantiations of the same base model, the nodes *not* modified by the mutator are considered identical. Retiarii can leverage such information to optimize the multi-model training (details in §5).

Mutator: an example. Figure 4 depicts a model space in which the third node ("model/maxpool") of the base model

```
1  # define the graph mutation behavior
2  class InceptionMutator(BaseMutator):
3    def __init__(self, paths_range, candidate_ops):
4      self.paths_range = paths_range # [2, 3, 4, 5]
5      self.ops = candidate_ops # {conv, dconv, ...}
6    def mutate(self, targets):
7      if not three_node_chain(targets):
8        return err
9      n = choose(candidates=self.paths_range)
10     delete_node(targets[1])
11     for i in range(n): # create n paths
12       op = choose(candidates=self.ops)
13       nd = create_node(name='way_'+str(i), op=op)
14       connect(src=targets[0].output, dst=nd.input)
15       connect(src=nd.output, dst=targets[2].input)
16
17 # mutation applied to the graph
18 apply_mutator(targets=["model/relu", "model/
       maxpool", "model/dense"],
19     mutator=InceptionMutator(
20     [2, 3, 4, 5], [conv, dconv, pool]))
```

Figure 6: A mutator that constructs an Inception-like cell.

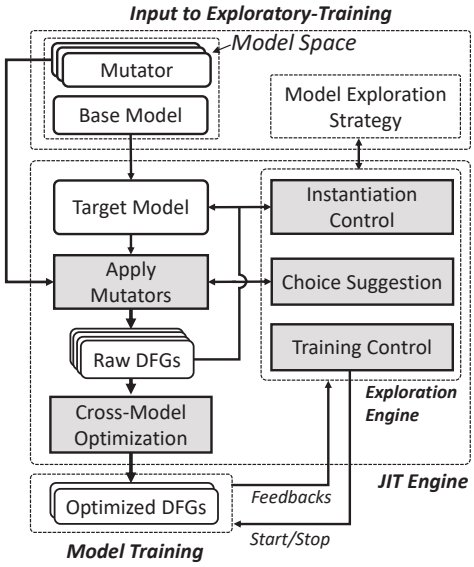


Figure 7: The architecture of Retiarii.

can be mutated with a multi-path cell. The cell could have 2 to 5 paths, each of which chooses an operator from *Conv*, *DepthwiseConv* and *Maxpool*. Figure 6 shows the code of the mutator, *i.e.,* `InceptionMutator`, which implements the model space illustrated in Figure 4.

All the mutation logic is encapsulated in the `mutate` function (lines 6-15). The entry point of the mutator is given by `targets` in the `mutate` function (line 6 of Figure 6), to match nodes/subgraphs in the given model. The `targets` of `InceptionMutator` is a chain of 3 nodes. This shows that a mutator can be applied to a subgraph with a specific pattern, which improves the reusability of a mutator. In the example code, the mutator first ensures that the matching is a chain of 3 nodes (lines 7-8). It will then call `choose` (line 9) to select an integer n to create n paths subsequently. On creating each path, the mutator will call `choose` (line 12) again to select an operator for the node in the path. Note that the code for a mutator can contain arbitrary complex control flow in a mutator (*e.g.,* the control loop in lines 11-15 of Figure 6), without polluting the instantiated models, unlike in the case of jumbo models with control flows. Finally, a call to `apply_mutator` will create a mutator instance (line 18), which matches a chain of *relu*, *maxpool*, and *dense*.

4 Retiarii Just-In-Time Engine

A key design decision for Retiarii to support exploratory-training is to instantiate models to explore on the fly and manage the training of instantiated models dynamically. This is accomplished by Retiarii's just-in-time (JIT) engine (Figure 7), which takes as input one or more base models, a set of mutators, and a policy describing the exploration strategy.

The end-to-end exploratory-training process is driven by the policy described as a *Model Exploration Strategy*. The JIT engine maintains a set of *target models*, initialized with the set of base models, and consults the model exploration strategy to decide which target model(s) and mutator(s) to choose (*i.e., Instantiation Control*), as well as which choices to make for each `choose` within those mutators (*i.e., Choice Suggestion*), to instantiate new models. The decision can be guided by a context-free strategy (*e.g.,* making a random choice upon each `choose`) or by a history-based strategy, generating choices based on which models have been previously instantiated [60]. The `choose` interface in Mutator enables the customization of the choices.

Once new models are instantiated (*i.e., Apply Mutators*) as Raw DFGs, the JIT engine transparently performs *Cross-Model Optimization* (§5). Because the JIT engine records the mutation history, the Cross-Model Optimization module can easily detect identical nodes across models to produce optimized DFGs by applying common sub-expression elimination [44], cross-model operator batching [15, 41], and NAS optimizations (§5). The optimized DFGs are then converted to the standard model format for the existing deep learning frameworks to perform single-model optimizations before training. In *Training Control*, the JIT engine launches training on new models, monitors the training of instantiated and optimized models, collects training feedbacks (*e.g.,* model accuracy), adjusts training priorities and resource allocation, and terminates training of less promising models, all guided by a model exploration strategy.

Retiarii's Mutator abstraction and JIT engine offers an elegant architecture to support exploratory-training, following the principles of separating policy from mechanisms and separation of concerns, and maximizing modularity, reusabil-

ity, and opportunities for optimizations. In addition to the common functionalities (*e.g.,* Cross-Model Optimization) in the overall infrastructure, mutators and policies encoding the model exploration strategies might also be reused. This is in sharp contrast to the current practice of encoding everything in a jumbo code, which is hardly understandable or reusable due to tight coupling.

5 Cross-Model Optimization

The DNN models instantiated by Retiarii in an exploratory-training process tend to have significant similarities as their DFGs share common subgraphs, thereby offering huge opportunities for Retiarii to optimize the training of multiple models. With mutators that identify and record all modifications to a model's DFG, Retiarii can easily find the common subgraphs of multiple DFGs, circumventing the generally NP-hard and APX-hard problem of identifying maximum common subgraphs [34].

5.1 Cross-Model Optimization Opportunities

Three different cross-model optimization opportunities are identified, depending on the inputs, weights, and trainability[2] of operators in the common subgraphs.

Common Sub-expression Elimination (CSE). CSE is a common compiler optimization to eliminate identical operations of a program by only computing them once. CSE can be applied to the non-trainable operators in the common subgraphs with common inputs and outputs, but cannot be applied to trainable operators as their weights will change during training, rendering their computation different after the first iteration. In practice, we find CSE particularly useful for merging prefix nodes of a DFG, because they are often non-trainable operators for data loading and preprocessing, as neural architecture search often uses the same dataset, batch size, and preprocessing procedures. When running multiple data-flow graphs concurrently on a single server, CSE can also avoid contention on shared storage and CPUs to maximize utilization of expensive GPUs.

Operator Batching. Common operators with different inputs and weights can potentially be batched together and computed in a single operator kernel. This optimization is useful for model exploration in multi-domain deep learning and transfer learning [28, 52, 53]. In this scenario, a model is modified to a new task with only minor changes, thus those modified models usually share a common skeleton. Adapter-based transfer learning is a one such example: networks have the same architecture from a pre-trained network, with adapters

[2]Similar to most popular deep learning frameworks, Retiarii allows model developers to specify whether the weights associated with an operator are *trainable*, whose weights will be applied with gradients during back-propagation.

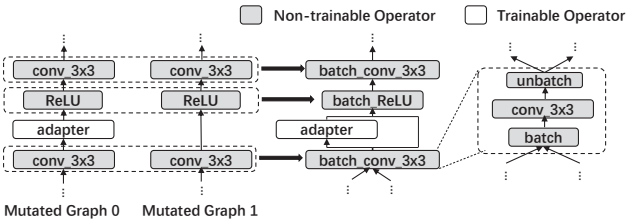


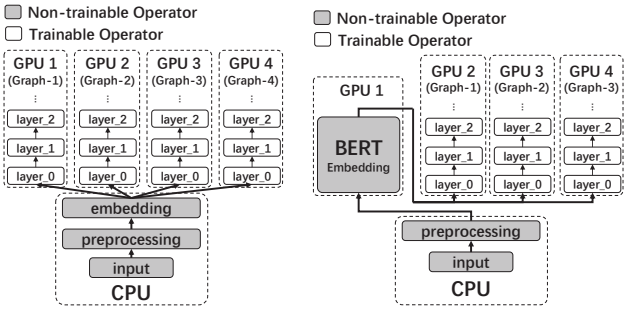
Figure 8: Operator batching: An example.

inserted at different locations. Only the inserted layers are fine-tuned [28, 52, 53]. Figure 8 illustrates an example that two graphs share multiple layers with the same weights. After merging the two graphs, the input of the common operators are batched along the batch dimension, and the output of the batched operators are split before adapters. Common operations with different weights (*e.g.,* trainable weights) can also be batched by leveraging special kernels (*e.g.,* grouped convolution [37], *batch_matmul*) that can parallelly compute on slices of an input tensor. This allows Retiarii to enable more fine-grained sharing of GPUs by increasing SIMD utilization with less GPU memory.

Super-Graph for Weight Sharing. Weight sharing is a machine-learning optimization shown to deliver improved empirical performance for certain model training: instead of training a graph's weights from scratch, shared weights are inherited from other graphs to continue the training in this graph [27, 50]. Retiarii naturally supports this training optimization by allowing model developers to annotate operator weights they want to share. Retiarii will automatically identify the weight sharing-enabled operators in common subgraphs. The DFGs with shared weights will be merged to build a super-graph. By training the super-graph together in one DFG, Retiarii can avoid overhead of checkpointing shared weights, because with weight sharing each graph has short training time (*e.g.,* several mini-batches). To accelerate the training of the merged super-graph, we further introduce a new type of parallelism when constructing executable graphs (§5.2) by increasing its scalability on distributed GPU clusters. Note that super-graphs are generated and used for optimizations only, and not exposed to developers.

5.2 Executable Graph Construction

To exploit these cross-model opportunities, Retiarii needs to construct executable graphs from the raw DFGs. The construction involves decisions of model merging, device placement of operators, and training parallelism, constrained by physical environment (*e.g.,* server configuration). Retiarii adopts a policy similar to Gandiva [64] that introspectively selects graphs to merge. Moreover, Retiarii specifically optimizes device placement of CSE-optimized graphs and training parallelism



(a) CPU-based embedding (b) GPU-based embedding

Figure 9: Device placement for CPU/GPU-based pre-trained embedding when constructing executable graphs.

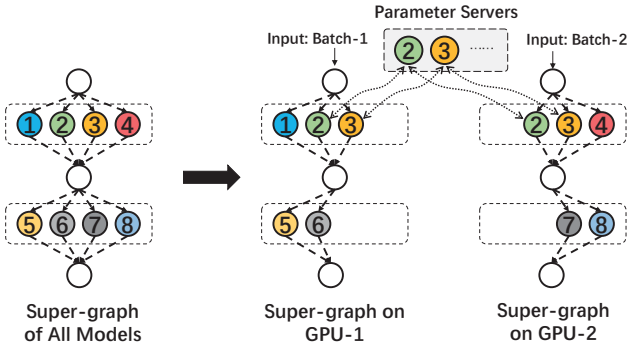


Figure 10: Retiarii uses mixed parallelism to improve scalability of weight sharing-based training.

of weight sharing.

Device Placement of CSE-Optimized Graphs For DFGs sharing the same dataset and preprocessing, these common operators can be merged by common sub-expression elimination. The most efficient execution plan of merged graphs depends on the types of merged operators and configuration of GPU servers. Figure 9 shows two different execution plans of CSE-optimized graphs. Both examples use a pre-trained embedding before the trainable layers. The difference is that the embedding in 9(a) is CPU-based (*e.g.,* word2vec [43]) while the embedding in 9(b) is GPU-based (*e.g.,* BERT [10, 19]). When BERT-embedding is the bottleneck of model computation and consumes most of GPU memory, dedicating one GPU for it can improve the pipeline and reduce memory consumption. Thus, we may pack more graphs on the rest of GPUs to improve the training throughput. Retiarii currently uses a whitelist to identify operators that require dedicated GPUs. We leave the automatic graph partitioning and optimization to future work.

In Retiarii, all cross-graph optimizations are applied within every batch of models. We first profiled the iteration time, peak GPU memory, and GPU utilization of each model by independently running for a few iterations. Then the models are sorted based on the iteration time. Retiarii greedily packs as many models as possible into one GPU. If the excutable graph's total training throughput is lower than that before optimization, the optimization will be reverted.

Mixed Parallelism for Weight Sharing. Weight sharing suffers from the scalability issue. After an exploration strategy instantiates a set of models, these models need to be trained sequentially (in an interleaved way) with different data to guarantee that every model can use the latest version of shared weights without losing training accuracy. A single model can hardly scale to a large number of GPUs using data parallelism, because a large batch size would harm model accuracy [25, 35]. Figure 10 shows an example of how Retiarii trains weight-shared models on two GPUs. Retiarii can improve the scalability by splitting the super-graph onto multiple GPUs, when the super-graph of all models is too large to fit into one GPU. Retiarii spreads the instantiated models into multiple super-graphs (each on a GPU) to be trained together. This can be regarded as model parallel training of the super-graph of all models. Moreover, in each iteration, models in different GPUs will be fed with different batches of data (like data parallelism), following the requirement of weight-shared training. The shared weights will be synchronously updated using parameter servers by averaging their gradients. Note that, it is difficult to apply Retiarii's mixed parallelism to a jumbo model, since a compiler can hardly understand and partition the sophisticated jumbo model without knowing each individual model's architecture. Our evaluation in §7.4 shows Retiarii's mixed parallelism yields better scalability that reduces the training time by up to 8.58× without affecting validation accuracy, compared with the traditional approach that trains the jumbo model using data parallelism.

6 Implementation

We have implemented Retiarii in about 19,723 lines of code, in which about 5,436 lines of code for the core Retiarii JIT engine, 5,203 lines of code for model, state, data management with failure recovery, and 9,084 for managing training with interfaces to various training services, such as Kubeflow [2]. We also wrote an additional 6,157 lines of code to implement 11 exploration strategies, 6 mutator classes, and 27 model spaces [4].

Building internal representations of base models and mutators. Our implementation supports base models defined in PyTorch and TensorFlow, which we convert to their graph representations. The conversion is done through TorchScript [9] for PyTorch. TensorFlow naturally supports a similar graph representation and offers the utility to output in a protobuf format. We do not yet support dynamic graphs. The mutators are extracted through Python Abstract Syntax Trees (AST) [1].

```
class ExplorationStrategy:
  # the APIs for instantiation control
  def generate_graph(self, new_graph_id)
  def on_ask_target_graph(self, graph_id)
  def on_ask_choice(self,graph_id,type,values,ctx)
  # the APIs for training control
  def execute_graph(self, graph_id, load_ckpt)
  def terminate_graph(self,graph_id,do_ckpt)
  def on_ask_training_approach(self, graph_id)
  # the APIs for getting provisioned information
  def query_mutation_history(self, graph_id)
  def on_receive_feedbacks(self,graph_id,feedback)
```

Figure 11: Some key APIs for an exploration strategy.

The base graph and mutators are then passed to the JIT engine.

Materializing the JIT engine. The JIT Engine drives the whole exploratory-training process. It first starts an exploration strategy which is an independent executable Python script. The strategy uses the APIs listed in Figure 11 to interact with the engine. Users are free to customize a new one following the interface. For instantiating a model, the mutators are applied one after another. On applying a mutator, the JIT engine retrieves the subgraphs specified by `targets`, and feeds them into the mutator. The instantiation is guided by an exploration strategy through those callback functions (*i.e.,* "on_*"). The JIT Engine maintains all the instantiated and trained models in a data store (*i.e.,* SQLite in our implementation) and collects runtime information of those models, such as model accuracy, execution time, which can by queried by the exploration strategy. Each model can have its training approach, *e.g.,* a training loop with a configured epoch number and batch size. We follow the practice in PyTorch Lightning [8] to provide a wrapper for programming and configuring a training approach. An exploration strategy can specify the training approach for each instantiated model.

Converting models for training. In our implementation, the optimized graphs are trained on current deep learning frameworks, such as TensorFlow and PyTorch. To make the optimized graph executable on those frameworks, we implement a converter to translate an optimized graph into TensorFlow or PyTorch code. Taking PyTorch as an example, the optimized graph is converted to a PyTorch module, *i.e.,* graph nodes in `__init__()` and connections in `forward()`. In cases where an optimized graph could contain multiple models, the losses are either added or concatenated to produce a single one. We enable device placement for a model with each framework's utility, such as the `to()` method in PyTorch and `with tf.device()` in TensorFlow.

Distributing exploratory-training. Exploratory-training usually requires lots of computation resources. In our implementation, Retiarii's JIT engine runs on a single machine, while the instantiated models can be distributed to wherever computing resources are available (*e.g.,* a cluster). For training of each model, Retiarii implements a wrapper to monitor its execution and collects metrics (*e.g.,* training performance) to report back to the JIT engine.

Tolerating and handling failures. As exploratory-training is usually time-consuming, in our implementation we deal with failures of both the JIT engine and model execution. All the exploration history is kept in the data store. When the JIT engine fails, it will be restarted and recover the state of exploration strategy by replaying the data in data store. For model exploration, the most valuable data are the set of models that have been explored and their observed results. These data are usually enough to continue an interrupted exploration from a previously known state. For an exploration strategy that maintains its own, additional states that cannot be recovered by our automatic mechanism, its own recovery logic must be provided. Another type of failure comes from the optimized graphs. If the execution of an optimized graph fails (*e.g.,* out of GPU memory, tensor shape mismatch), while each model in this graph runs without error, Retiarii will revert to running the individual models separately.

Limitations. Retiarii has limited support to dynamic graphs. Retiarii's mutators are applied to a base model. However, sometimes it is difficult to extract a graph representation from the a highly dynamic base model (*e.g.,* Tree-LSTMs [58]). Also, the current implementation of operator batching is limited. Some operator batching is possible but is not implemented as it requires implementing new GPU kernels. Moreover, when a model is mutated, it requires additional programming efforts to match the shape of adjacent layers' input/output tensors. It is currently out of the scope of Retiarii to handle possible shape mismatch after mutation. We leave automatic shape inference and matching to our future work.

7 Evaluation

We evaluate the performance of Retiarii for exploring neural network architectures. Overall, the key findings include:

- The separation of model space and exploration strategy makes it easy for Retiarii to try different combinations. Retiarii currently supports 27 popular Neural Architecture Search (NAS) solutions. Most of them can be implemented by the three mutator classes provided by Retiarii.
- A number of micro-benchmarks show how Retiarii's cross-model optimizations greatly improve training efficiency.
- Retiarii improves the model exploration speed of three NAS solutions by up to 2.58×, compared with traditional approaches.
- Retiarii improves the scalability of weight sharing-based NAS solutions and brings up to 8.58× speed-up using the proposed mixed parallelism, compared with data parallelism.

NAS Solution	Model Space	Exploration Strategy	Required Mutator Class			
			Input Mutator	*Operator Mutator*	*Inserting Mutator*	*Customized Mutator*
MnasNet [59]	MobileNetV2-based space	Reinforcement Learning		✓	✓	
NASNet [70]	NASNet cell	Reinforcement Learning	✓	✓		
ENAS-CNN [50]	NASNet cell variant	Reinforcement Learning	✓	✓		
AmoebaNet [51]	NASNet cell	Evolutionary	✓	✓		
Single-Path One Shot (SPOS) [27]	ShuffleNetV2-based space	Evolutionary		✓		
Weight Agnostic Networks [23]	Evolving space w/ adding/altering nodes adding connections	Evolutionary		✓		✓
Path-level NAS [13]	Evolving space w/ replication and split	Reinforcement Learning				✓
TextNAS [62]	TextNAS space	Reinforcement Learning	✓	✓		
...	...	...	...	...	...	...

Table 1: NAS solutions currently supported by Retiarii, and required mutators to implement them in Retiarii. Please refer to [4] for the full list that contains 27 NAS solutions in total.

7.1 Expressiveness and Reusability

Table 1 shows 8 out of 27 NAS solutions currently supported by Retiarii (please refer to [4] for the full list). After decoupling model spaces from exploration strategies, developers can easily reuse them without extra coding efforts. For example, the exploration strategy "reinforcement learning" is reused by MnasNet [59], NASNet [70] and ENAS-CNN [50]. Several machine learning researchers at Microsoft Research are now using Retiarii to explore more NAS solutions by leveraging different combinations of model spaces and exploration strategies.

To build these model spaces, Retiarii provides three default mutator classes. *Input Mutator* is to mutate inputs of matched operators. *Operator Mutator* is to replace matched operators with other operators. *Inserting Mutator* is to insert new operators or sub-graphs after matched operators. We find the three mutator classes are enough to implement most of the listed NAS solutions. Moreover, Retiarii allows model developers to build customized mutator classes using basic graph mutation primitives to implement more complex model spaces, *e.g.,* Weight Agnostic Networks [23] and Path-level NAS [13].

7.2 Micro-benchmarks

7.2.1 Shared Data Loading and Preprocessing

The following experiments demonstrate two micro-benchmarks of common sub-expression elimination, where multiple models share the same data loading and preprocessing. These micro-benchmarks are evaluated on 4 V100 GPUs with 20 CPU cores. We compare Retiarii with a baseline that runs each model independently without CSE. For a fair comparision, CUDA Multi-Process Service (MPS) [5] is enabled for the baseline when Retiarii decides to packed more than one model in a GPU.

Avoiding CPU Bottleneck. Figure 12 shows the aggregate throughput and CPU usage with the increased number of MnasNet0.5 (*i.e.,* depth multiplier=0.5) models [59] running concurrently on the 4 V100 GPUs and 20 CPU cores. The models are trained on ImageNet with a batch size of 224 with the same preprocessing as in [59]. The baseline approach runs each model independently, thus each batch of data will be loaded and preprocessed multiple times. Retiarii merges the data loading and preprocessing across different models thus they are executed only once. Both Retiarii and the baseline can further pack multiple models into one GPU to run them concurrently. The models are distributed in a round-robin way. For example, when running 6 models, the first two GPUs are packed with two models on each GPU, while each of rest two GPUs runs only one model. The measured performance is averaged over one training epoch.

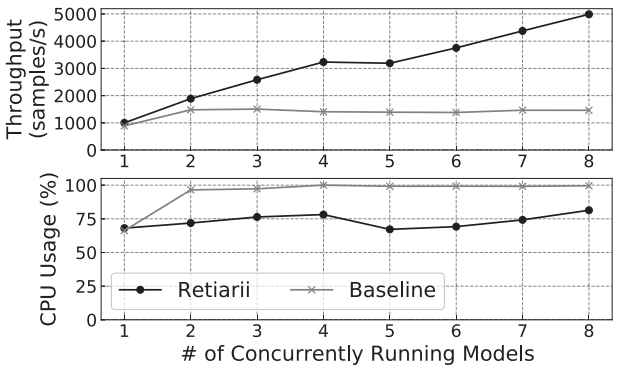


Figure 12: The aggregate throughput and CPU usage with varying number of concurrently running MnasNet0.5 models.

Overall, Retiarii increases the throughput by 3.41× when running 8 models on 4 GPUs. The bottom figure in Figure 12 shows that training one MnasNet0.5 model has already consumed about 75% of CPU cores. Thus, CPU will become

the bottleneck when running more than one model without sharing. On the contrary, Retiarii eliminates the redundant data loading and preprocessing. Increasing the number of concurrent models does not affect the CPU usage for data loader. The marginal increase of CPU usage in Retiarii is due to other computations, which can not be merged (*e.g.,* overhead of the training runtime).

Also note that, running 5 models does not bring higher throughput than running 4 models. This is due to the overhead of synchronization brought by unbalanced model assignment, *i.e.,* the first GPU is packed with two models while each of the rest three GPUs runs only one model. In Retiarii, merging the graphs will force them to be trained synchronously. Packing two models in one GPU may increase their iteration time, thus the rest three GPUs have to wait for the two slower models in the first GPU in every iteration. This suggests to merge the graphs with a similar iteration time to avoid severe synchronization overheads.

Avoiding GPU Bottleneck. Non-trainable embedding [49] can be regarded as a special type of data preprocessing. In this micro-benchmark, we use BERT [19] to obtain pre-trained contextual embeddings of input tokens from Stanford Sentiment Treebank (SST) dataset [55] for training TextNAS [62], which is one of the state-of-the-art natural language processing models. The batch size for each TextNAS model is 128. Different from the micro-benchmark of avoiding CPU bottleneck, the embedding computation is placed on GPU because the BERT embedding runs much faster on GPU than CPU [3]. The baseline still runs multiple models independently. The performance is measured by averaging over one training epoch.

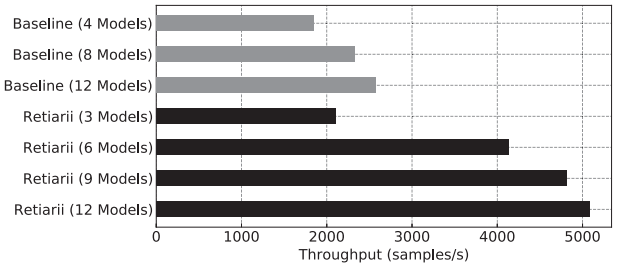


Figure 13: The aggregate throughput with varying number of TextNAS models.

Figure 13 shows the result. Overall, Retiarii achieves 1.97× throughput of the baseline when training 12 TextNAS models on 4 V100 GPUs. Both the baseline and Retiarii meet out-of-memory when running more than 12 TextNAS models. As we have shown in Figure 9, Retiarii uses model parallelism to dedicate one GPU to compute the BERT embedding, which is pipelined with the training of TextNAS models on the other three GPUs. Since the BERT embedding is the bottleneck in each training iteration, this optimization allows the training of more TextNAS models to be overlapped with the BERT

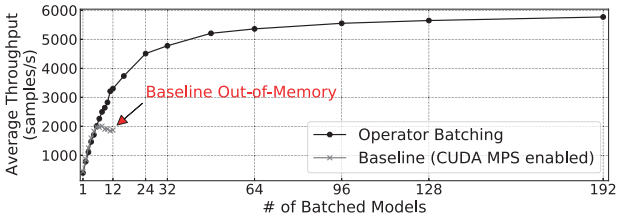


Figure 14: The aggregate throughput with varying number of batched models.

embedding. In this experiment, we find Retiarii can pack two TextNAS models on each GPU (*i.e.,* 6 models in total) without affecting the iteration time. And 12 models can be packed in total with better aggregated throughput, but each model's iteration time is degraded. Although the baseline can also pack up to 12 models on 4 GPUs, the compute-intensive BERT embedding repeats three times per GPU that greatly increases the iteration time. Only marginal improvement on throughput is observed in the baseline when packing more models using CUDA MPS.

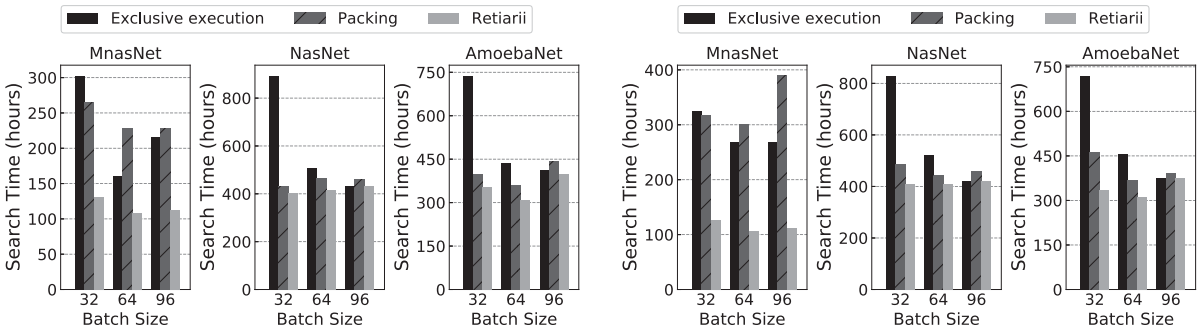
7.2.2 Operator Batching

To evaluate operator batching across graphs, we insert adapter layers to a pre-trained MobileNet [29]. Weights from the MobileNet are fixed during training. These models use the same batch size, which is 8 images per mini-batch. Synthetic data without preprocessing is used to avoid the gain from shared data loading. The models are trained on one V100 GPU of 16GB GPU memory. Similar to previous micro-benchmarks, the baseline uses CUDA MPS to execute multiple models in one GPU. The performance is measured by averaging the throughput over 1500 mini-batches.

Figure 14 shows the average throughput of concurrently running models. Overall, Retiarii's operator batching improves the aggregate throughput by 3.08× when batching 192 models, compared with the baseline that can only train at most 12 models together. Retiarii can batch more models than the baseline because it only has one copy of (fixed) weights from MobileNet. Only the memory for adapters is increased when batching more models. Even when Retiarii batches 12 models, it still achieves 1.76× improvement on the aggregate throughput. This improvement comes from the benefit of vectorization to execute the batched operators in a single kernel, which increases GPU utilization.

7.2.3 Optimization for Weight Sharing

To evaluate Retiarii's optimization for weight sharing, we use Single Path One-Shot (SPOS) [27] to explore a model space built by ShuffleNetV2 blocks, where a model is instantiated for every batch of data. The models are trained with



(a) NVIDIA Data Loading Library (DALI)

(b) PyTorch DataLoader

Figure 16: The completion time of the search phase of three NAS approaches, each of which generates 1,000 models for training.

synthetic data on a V100 GPU of 16GB GPU memory. We implemented two baselines that share weights of overlapped operators among the instantiated models through weight saving and loading. In the first baseline, a checkpoint file is used for weight sharing, *i.e.,* a model loads its weights from the file, then saves its updated weights to the file after training a mini-batch. In the second baseline, the file is replaced with a `dict` object located in GPU memory. Both model weights and optimizer states (*e.g.,* momentum) need to be checkpointed.

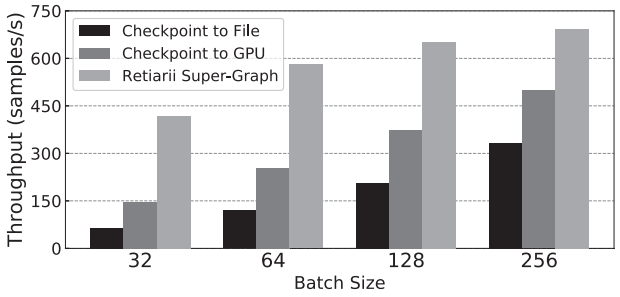


Figure 15: The throughput of weight sharing with and without cross-model optimization.

The result is shown in Figure 15. By merging multiple models as a super-graph, Retiarii's cross-model optimization improves the throughput by up to 6.52× when batch size is 32, and 2.08× when batch size is 256 (compared with checkpoint-to-file). Since SPOS only trains an instantiated model with a batch of data, frequent checkpointing brings significant overheads. Merging instantiated models into a super-graph allows Retiarii to load the models only once (at the beginning). Thus, Retiarii can use control flow to only activate the desired model, which also saves the time of model initialization. The performance of a jumbo model is similar to that of Retiarii's super-graph, the difference is that the super-graph is automatically built by Retiarii's JIT engine while the same graph is manually programmed in the jumbo model approach. This leads to a big performance gap on parallel training which will be illustrated in §7.4, as Retiarii fully understands each sampled graph and the weight sharing pattern.

7.3 Speeding up Neural Architecture Search

To evaluate the performance of running NAS solutions on Retiarii, we select three popular and representative NAS solutions from Table 1: (1) MnasNet [59], (2) NASNet [70], and (3) AmoebaNet [51]. They cover different combinations of model spaces and exploration strategies: MnasNet and NASNet use the same search strategy; NASNet and AmoebaNet have the same model space. We compare Retiarii against the one-off solutions built by traditional deep learning frameworks. Since Retiarii separates the cross-model optimization from model generation, all the three NAS solutions can leverage the three cross-model optimizations in §5 to accelerate the search of architectures without extra effort.

To evaluate the traditional approaches, which are unaware of cross-model relations, we test the following two baselines. (1) **Exclusive execution**: a model is trained independently and exclusively on one GPU. (2) **Packing**: multiple models may share the same GPU using CUDA MPS without merging their graphs; it uses Retiarii's decisions to choose the models to be packed onto the same GPU.

In the experiments, each NAS approach will generate 1,000 models in 20 batches (each batch contains 50 models). For a fair comparision, Retiarii and the two baselines are given the same set of models in the same order. We compare the time to finish the training of all the 1,000 models for 1 epoch on ImageNet's training images [18]. We vary the batch size from 32 to 96 (batch size larger than 96 will lead to Out-of-Memory). We also compare the performance using two implementations of the data loader, *i.e.,* NVIDIA Data Loading Library (DALI) [6] and PyTorch DataLoader [7], to understand the impact of data loading. The experiments are conducted on 4

NVIDIA Tesla V100 GPUs (each has 16GB GPU memory). Simlar to [64], we "fast-forward" the experiment by instructing model trainings to skip a number of iterations when the iteration time becomes stable. We measure the average iteration time over 100 mini-batches, which is used to project the entire training time.

Figure 16 shows the results of the search time of different settings. In most settings, Retiarii is substantially faster than the two baselines due to the cross-model optimizations. Overall, Retiarii achieves up to 2.57× speed-up on the search time compared with the two baselines. The performance gain mainly comes from packing multiple models in the same GPU, and shared dataloading and preprocessing. Because the packing baseline is agnostic to the cross-model relations, it cannot apply cross-model optimizations thus only brings up to 1.42× speed-up over the exclusive execution. Moreover, due to the increased CPU burden on the larger batch size, the packing baseline runs even slower by 31% than the exclusive execution on PyTorch DataLoader when the batch size is 96. Note that, an introspective policy, *e.g.,* Gandiva [64], can remedy the packing baseline's slow-down by reverting the packing when the training speed is slower. But the key insight in this experiment is that only using packing without cross-model merging will limit the space for improvement.

Retiarii achieves higher speed-up on MnasNet than NASNet and AmoebaNet. Because the models from MnasNet are designed for mobile devices that have a lower GPU memory usage and shorter iteration time, Retiarii can pack more MnasNet models into one GPU and merge their graphs for cross-model optimizations. As the generated models have different memory consumption, the number of models that can be fit in the same GPU varies accordingly. When the batch size is 32, Retiarii can run 4-22 MnasNet models simutanously; but only 4-8 NASNet/AmoebaNet models due to the larger model size. We also observe Retiarii achieves higher speed-up on PyTorch DataLoader, because DALI is more efficient on data preprocessing that reduces the probability of bottleneck on CPU.

7.4 Scaling Weight-Shared Training

In addition to system optimizations, Retiarii also enables and enhances the weight sharing optimization advocated by the machine learning community. As shown in §7.2.3, Retiarii builds a super-graph for weight sharing to avoid the overhead of model building and checkpointing. This optimization can be further improved by training the super-graph using mixed parallelism to scale it to a GPU cluster.

In this experiment, we build a model space with ShuffleNetV2 blocks described in the Single Path One-Shot (SPOS) paper [27]. Each model in the model space is randomly sampled and trained for one batch of data [17,27]. The models are trained for 60 epochs in total on the ImageNet dataset (with 1,281,167 images). As a result, a new model is instantiated for every batch of data (*e.g.,* $1281167/256 \times 60 = 300240$ models are instantiated when the batch size is 256). The experiment runs on two servers, each has 4 V100 GPUs. We use the common evaluation metric of weight sharing-based approaches [12, 27] to evaluate the searched space. We randomly sample 196 models and evaluate each model using 256 images from ImageNet's validation set. Then we calculate the average validation accuracy of the 196 models. The higher the average validation accuracy is, the better the space is explored. We compare Retiarii's mixed parallelism with three commonly used data parallelism approaches. To understand the benefit of mixed parallelism, all the three baselines of data parallelism and Retiarii's mixed parallelism enable the super-graph optimization (*i.e.,* no saving and loading of weights). Specifically, the former three are manually programmed jumbo-models, while the latter is a super-graph automatically built by Retiarii.

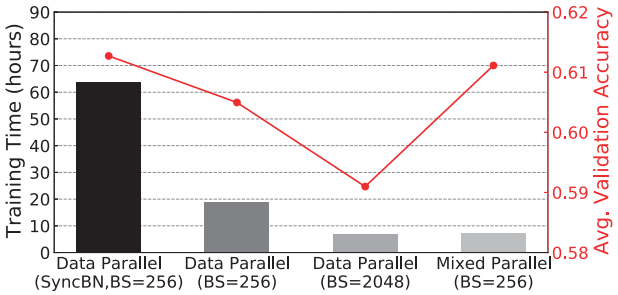


Figure 17: Training time and validation accuracy of weight sharing. The left y-axis shows the training time (bar chart). The right y-axis shows the validation accuracy (line chart).

Figure 17 shows the training time and validation accuracy of the three data parallelism approaches and Retiarii's mixed parallelism. The data parallelism of the left two bars and Retiarii's mixed parallelism use the batch size of 256 with a learning rate of 0.125 per model (or per 256 data samples). As a common practice of data parallelism, scaling to 8 GPUs requires to split each batch of data across the 8 GPUs (*i.e.,* the batch size per GPU is 32). SyncBN [66] is an optimization to calculate batch normalization across multiple GPUs, which proves to improve the model quality, but slows down the training due to intensive synchronization and data transmission across GPUs. As shown in Figure 17, SyncBN-based data parallelism requires more than 60 hours of training time. Disabling SyncBN reduces the training time to $\sim$ 20 hours but harms the model accuracy. In contrast, Retiarii's mixed parallelism greatly reduces training time (only 7.45 hours), achieving up to 8.58× speed-up over SyncBN-based Data Parallel training. This is because the mixed parallel training avoids the synchronization overhead of SyncBN as each GPU runs a different model requiring no cross-GPU synchroniza-

tion. Moreover, Retiarii's mixed parallel training produces a comparable validation accuracy to SyncBN-based Data Parallel training (61.49% v.s. 61.11%). Another practice of data parallelism is to increase batch size and learning rate with the increased number of GPUs. The result is shown as the second bar on the right of Figure 17. Although the training time is reduces to 7.04 hours, the model's validation accuracy degrades significantly. This result is consistent with the common wisdom in deep learning community that larger batch size could hurt model accuracy [25, 35]. In summary, Retiarii's mixed parallelism achieves better scalability for weight-shared training, without sacrificing model accuracy.

8 Related Work

Deep Learning Frameworks. Deep learning frameworks (*e.g.,* PyTorch [48], TensorFlow [11], and MXNet [14]) are designed to describe and train an individual DNN model, which covers only one step in the end-to-end exploration-training process of devising a high-quality model.

Network Architecture Search Algorithms. To automate the design of neural networks, Neural Architecture Search (NAS) [39, 50, 59, 60, 69, 70] develops algorithms to discover the state-of-the-art neural model architecture. Limited by the existing deep learning frameworks, their implementations often couple model space, exploration strategy, and model training together, introducing barriers to innovations and optimizations. In contrast, Retiarii's modular and decoupled approach maximizes reusability and facilitates optimizations.

AutoML Systems. Automated Machine Learning (AutoML) automates the end-to-end process of real-world machine learning problems, *e.g.,* AutoGluon [21], TPOT [47], Auto-Sklearn [22], Auto-WEKA [61], AutoKeras [32]. The implementations of these systems still couple the domain-specific model space and exploration strategy, making it hardly reusable to other problem domains.

The hyper-parameter tuning systems like Google Vizier [24] and Katib [68] can be used for neural architecture search. To use a hyper-parameter tuning system, the model space and exploration strategy are being parameterized. Since different model space and exploration strategy often use a different set of parameters, this approach limits the reusability of the implementation. Moreover, the hyper-parameter tuning approach can limit the expressiveness of the system as well. Some model space is hard to be parameterized, *e.g.,* evolutionary NAS [13, 23, 51]. It is worth noting that Retiarii's Mutator abstraction can also be used for hyper-parameter tuning. The hyper-parameter tuning can be treated as a special case of neural model mutation.

DeepArchitect [46] also strives to decouple model spaces and exploration strategies. Compared to DeepArchitect, Retiarii differentiates itself with the Mutator abstraction. As shown in §7, Retiarii can implement multiple model spaces using a few mutators, demonstrating great reusability and composability. More importantly, with the Mutator abstraction, Retiarii is able to exploit cross-model optimizations easily, which is not addressed previously.

Graph Optimization for Deep Learning. Recently, there are many proposals to optimize the computation of a single neural network model by optimizing the data-flow graph, *e.g.,* TVM [15], DLVM [63], TensorFlow-XLA [38], TASO [31], TensorFlow-Fold [41]. In contrast, Retiarii exploits the cross-model optimizations exposed by Mutator. HiveMind [45], FLEET [26] and some other works [40] apply common sub-expression elimination in the AutoML scenario to deduplicate the common prefix nodes of multiple graphs. This can be considered a special case in Retiarii's larger optimization space, which includes other techniques like operator batching and weight sharing.

9 Conclusion

We propose Retiarii, the first deep learning framework that supports the exploratory training on a neural network model space, rather than on a single neural network model. The core of Retiarii is the Mutator abstraction, which not only allows the specification of different neural network model spaces, interacts with various model exploration strategies, and exposes the model correlations for further optimization, but also serves as a clean interface to separate the three. The design leads to ease of programming, reuse of model space, exploration strategy, and cross-model optimization. Our evaluation demonstrates the effectiveness of the design, showing more than 8× improvement in the overall exploratory-training performance over approaches that rely on existing deep learning frameworks, which only support one model at a time. The artifacts of Retiarii are available at `https://github.com/microsoft/nni/tree/retiarii_artifact`.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof. Byung-Gon Chun, for their extensive suggestions. We thank Jim Jernigan and Kendall Martin from the Microsoft Grand Central Resources team for providing GPUs for the evaluation of Retiarii. We also thank our colleagues at Microsoft, for their help in implementing and deploying Retiarii: Chengmin Chi (STCA), Shinai Yang (STCA), Deshui Yu (STCA), Chuanjie Liu (STCA). Fan Yang thanks the late Pearl, his beloved cat, for her faithful companion during writing this paper.

References

[1] ast – Abstract Syntax Trees. `https://docs.python.org/3/library/ast.html`, 2020. Online; accessed 30 April 2020.

[2] Kubeflow, The Machine Learning Toolkit for Kubernetes. https://www.kubeflow.org/, 2020. Online; accessed 30 April 2020.

[3] Microsoft open sources breakthrough optimizations for transformer inference on GPU and CPU. https://bit.ly/2xBD70N, 2020. Online; accessed 30 April 2020.

[4] Nas solutions supported by retiarii. https://github.com/microsoft/nni/blob/retiarii_artifact/nas_allstar.md, 2020.

[5] NVIDIA CUDA Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020. Online; accessed 30 April 2020.

[6] NVIDIA DALI documentation. https://docs.nvidia.com/deeplearning/sdk/dali-developer-guide/index.html, 2020. Online; accessed 30 April 2020.

[7] PyTorch DataLoader. https://pytorch.org/docs/stable/data.html, 2020. Online; accessed 30 April 2020.

[8] The lightweight PyTorch wrapper for ML researchers. https://github.com/PyTorchLightning/pytorch-lightning, 2020. Online; accessed 30 April 2020.

[9] TORCHSCRIPT. https://pytorch.org/docs/stable/jit.html, 2020. Online; accessed 30 April 2020.

[10] Using BERT to extract fixed feature vectors (like ELMo). https://github.com/google-research/bert, 2020. Online; accessed 30 April 2020.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[12] Gabriel Bender. Understanding and simplifying one-shot architecture search. 2019.

[13] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. *arXiv preprint arXiv:1806.02639*, 2018.

[14] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[16] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[17] Xiangxiang Chu, Bo Zhang, Jixiang Li, Qingyuan Li, and Ruijun Xu. Scarletnas: Bridging the gap between scalability and fairness in neural architecture search. *arXiv preprint arXiv:1908.06022*, 2019.

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE CVPR 2009*.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[21] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

[22] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*, pages 113–134. Springer, 2019.

[23] Adam Gaier and David Ha. Weight agnostic neural networks. In *Advances in Neural Information Processing Systems*, pages 5365–5379, 2019.

[24] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

[25] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[26] Hui Guan, Laxmikant Kishor Mokadam, Xipeng Shen, Seung-Hwan Lim, and Robert Patton. FLEET: Flexible efficient ensemble training for heterogeneous deep neural networks. *MLSys 2020*, 2020.

[27] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*, 2019.

[28] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. *arXiv preprint arXiv:1902.00751*, 2019.

[29] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[30] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[31] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[32] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019.

[33] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pages 2016–2025, 2018.

[34] Viggo Kann. On the approximability of the maximum common subgraph problem. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 375–388. Springer, 1992.

[35] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[36] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[38] Chris Leary and Todd Wang. XLA: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

[39] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[40] Rui Liu, Sanjan Krishnan, Aaron J Elmore, and Michael J Franklin. Understanding and optimizing packed neural network training for hyper-parameter tuning. *arXiv preprint arXiv:2002.02885*, 2020.

[41] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[42] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018.

[43] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[44] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[45] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating deep learning workloads through efficient multi-model execution. In *NIPS Workshop on Systems for Machine Learning (December 2018)*, 2018.

[46] Renato Negrinho, Matthew Gormley, Geoffrey J Gordon, Darshan Patil, Nghia Le, and Daniel Ferreira. Towards modular and programmable architecture search. In *Advances in Neural Information Processing Systems*, pages 13715–13725, 2019.

[47] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*, pages 151–160. Springer, 2019.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al.

PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[49] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[50] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268, 2018.

[51] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[52] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. Efficient parametrization of multi-domain deep neural networks. In *IEEE CVPR 2018*.

[53] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. Learning multiple visual domains with residual adapters. In *Advances in Neural Information Processing Systems*, pages 506–516, 2017.

[54] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE CVPR 2018*.

[55] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[56] Francisco J Solis and Roger J-B Wets. Minimization by random search techniques. *Mathematics of operations research*, 6(1):19–30, 1981.

[57] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *IEEE CVPR 2016*.

[58] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[59] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018.

[60] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[61] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[62] Yujing Wang, Yaming Yang, Yiren Chen, Jing Bai, Ce Zhang, Guinan Su, Xiaoyu Kou, Yunhai Tong, Mao Yang, and Lidong Zhou. Textnas: A neural architecture search space tailored for text representation. *arXiv preprint arXiv:1912.10729*, 2019.

[63] Richard Wei, Lane Schwartz, and Vikram Adve. DLVM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

[64] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[65] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

[66] Hang Zhang, Kristin Dana, Jianping Shi, Zhongyue Zhang, Xiaogang Wang, Ambrish Tyagi, and Amit Agrawal. Context encoding for semantic segmentation. In *IEEE CVPR 2018*.

[67] Hongpeng Zhou, Minghao Yang, Jun Wang, and Wei Pan. Bayesnas: A bayesian approach for neural architecture search. *arXiv preprint arXiv:1905.04919*, 2019.

[68] Jinan Zhou, Andrey Velichkevich, Kirill Prosvirov, Anubhav Garg, Yuji Oshima, and Debo Dutta. Katib: A distributed general automl platform on kubernetes. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, pages 55–57, 2019.

[69] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[70] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.

A Artifact Appendix

A.1 Abstract

This artifact is designed to reproduce the main results of this work, which have two goals:

- **Functionality**: Retiarii can express NAS spaces using mutators, explore spaces using Exploration Engine, and accelerate the exploration using cross-model optimization.
- **Performance**: Retiarii's cross-model optimization achieves the performance number claimed in §7.

A.2 Artifact check-list

- **Algorithm:** yes
- **Data set:** ImageNet [18], SST [55]
- **Run-time environment:** Ubuntu 16.04, CUDA 10.0, cuDNN 7.6.5. Root access is required.
- **Hardware:** GPUs with NVIDIA MPS.
- **Metrics:** training throughput; job completion time; model validation accuracy.
- **Output:** Web UI; stdout from console.
- **Required disk space:** 200 GB
- **Expected experiment run time:** 20 hours
- **Public link:** `https://github.com/microsoft/nni/tree/retiarii_artifact`
- **Code licenses:** MIT License

A.3 Description

A.3.1 How to access

Clone the "retiarii_artifact" branch of Microsoft NNI's GitHub repository.

```
git clone -b retiarii_artifact https://github.com/Microsoft/nni.git
```

A.3.2 Hardware dependencies

This artifact requires at least one server with four NVIDIA V100 GPUs.

A.3.3 Software dependencies

- CUDA 10.1;
- cuDNN 7.6.5;
- Python 3.7;
- NVIDIA DALI;
- NVIDIA Apex;
- PyTorch 1.5.1;
- TensorFlow 2.3;
- Other Python packages in "requirements.txt".

A.3.4 Data sets

- ImageNet: should be placed at "retiarii_perf/data/imagenet".
- SST: The three text files (dev.txt, test.txt, train.txt) SST should be placed at "retiarii_perf/data/sst/trees".

A.4 Installation

For running Retiarii's artifact, please install NNI v1.8 first. This artifact contains two parts. In the folder of "retiarii_nas", we demonstrate the functionality of Retiarii to express different NAS solutions. In the folder of "retiarii_perf", we evaluate Retiarii's performance using cross-model optimization.

For some experiments, it requires NVIDIA MPS to be enabled. To start NVIDIA MPS:

```
sudo ./mps_scripts/init_mps_for_all_gpus.sh
./mps_scripts/set_mps_env_for_all_gpus.sh
```

To stop NVIDIA MPS:

```
sudo ./mps_scripts/stop_mps_for_all_gpus.sh
```

A.5 Evaluation: NAS Solution All-stars

In the folder of "retiarii_nas", we have implemented 17 NAS solutions using Retiarii. We support both PyTorch and TensorFlow. Weight Agnostic Networks (wann), Path-level NAS (path_level), and Hierarchical Representation (hierarchical) are implemented with TensorFlow. Other NAS solutions are implemented with PyTorch. We also provide a script to test them, which can be started using the following command.

```
python3 retiarii.py e2e_launch.py [nas_mode]
```

(Use "python3 retiarii.py -L" to get the list of supported models)

After the command is executed, a Web UI URL will be given, which contains the trial execution status.

Note that, to speed up the test, we run each generated model by only one mini-batch (thus, returned values are all 0), you are free to remove the 'break's in e2e_launch.py (ModelTrain, ModelTrainCifar, ModelTrainTextNAS) to run each generated model completely.

This artifact has supported three classic exploration strategies: random, reinforcement learning, and evolution, and also has supported two differentiable training strategies: DARTS training strategy and ProxylessNAS training strategy. Other exploration strategies have been supported in NNI (`https://github.com/microsoft/nni/blob/retiarii_artifact/backend_nni/docs/en_US/Tuner/BuiltinTuner.md`), have not been integrated into this artifact. They will be formally supported in Retiarii official release.

Paper Claim: Retiarii is able to support 27 NAS solutions.

Clarification: We have included 17 of the 27 NAS solutions in the artifact evaluation. The remaining ones only have minor differences with the included implementations (e.g., EfficientNet v.s. MnasNet, SCARLET-NAS v.s. FairNAS v.s. SPOS). We believe the included ones are sufficient to demonstrate the programmability of Retiarii. Full support of the 27 NAS solutions will be included in an official release version of Microsoft NNI.

A.6 Evaluation: Retiarii Performance

A.6.1 Micro-benchmark: Deduplication to avoid CPU bottleneck

Execution command:

```
python artifact_start.py micro_dedup_cpu --n=8
```

This python script will start 8 jobs (each GPU runs two jobs), then profile the total throughput. This command takes 1.5 minutes. The result will be printed after the profiling as follows. The error should be within 10%.

```
Throughput: 4746.849792184445 samples/s
```

Paper Claim: In Figure 12, when running 8 models, Retiarii can achieve about 5000 samples/s.

A.6.2 Micro-benchmark: Deduplication to avoid GPU bottleneck

Execution command:

```
python artifact_start.py micro_dedup_gpu --n=12
```

This python script will start 12 jobs. GPU-0 runs one job, each of the other three GPUs run 4 jobs). Then it profiles the total throughput. This command takes 1.5 minutes. The result will be printed after the profiling as follows. The error should be within 10%.

```
Throughput: 5028.187640607402 samples/s
```

Paper Claim: In Figure 13, when running 12 models, Retiarii can achieve about 5100 samples/s.

A.6.3 Micro-benchmark: Operator batching

Execution command:

```
python artifact_start.py micro_batching --n=192
```

This python script will use Retiarii to pack 192 models into one job to be run on a single GPU-0. Then it profiles the total throughput. This command takes 10 minutes. It is normal if it has no output for a long time, because it takes 3 minutes for the cross-model optimization policy to calculate a plan. The result will be printed after the profiling as follows. The error should be within 10%.

```
Throughput: 6124.981150684514 samples/s
```

Paper Claim: In Figure 14, when batching 192 models, Retiarii can achieve about 5800 samples/s.

A.6.4 End-to-end Evaluation: MnasNet using DALI

Execution command:

```
python artifact_start.py e2e_dali_mnasnet
```

This experiment will train 1000 MnasNet models in 20 batches (each batch has 50 models). Each model will be trained for 1 epoch on ImageNet, which will be very time-consuming and costly if we train all 1000 models. Since we only want to know the training time but not the validation accuracy. We use a workaround to "fast-forward" the training. We profile each job for 150 mini-batches to measure the iteration time. Then we use the measured job speed to emulate the experiment with a simple job scheduler (implemented in "fast_scheduler.py"). The experiment takes about 1 hour to run. The result will be printed as follows. The error should be within 10%.

```
124.35633072276445 hours for mnasnet w/ BS=32
```

Paper Claim: In Figure 15(a), when Batch Size=32, Retiarii can finish NAS exploration of MnasNet in about 130 hours.

A.6.5 End-to-end Evaluation: SPOS training using mixed parallelism

Execution command:

```
python artifact_start.py e2e_spos_mix_parallel
  --n=4
```

This python script will start 4 jobs, each on one GPU, to train SPOS in mixed parallelism, a new type of training parallelism we proposed for weight sharing-based training. The super-graph is generated in the function "_gen_spos_super_graph(n_job)" in "artifact_start.py". In the paper, we used 8 V100 GPUs in two servers, which takes about 7.45 hours to train SPOS for 60 epochs achieving 61.2% average validation accuracy. The result will be printed as follows.

```
[03/31 02:40:46] INFO (main) Epoch [60/60]
    Validation Step [196/196]  acc1 0.650000
    (0.611117)  acc5 0.887500 (0.833490)  loss
    2.359303 (2.586974)
```

Note that, the training of SPOS is unstable. The average validation accuracy could vary from 60% to 62%. For your reference, we also provide the training log we obtained on eight V100 GPUs in "data/spos_8_v100.log".

Paper Claim: In Figure Figure 17, Retiarii's mixed parallelism can train SPOS for 60 epochs with a batch size of 256 to achieve 61.11%.

A.7 Experiment customization

New experiments can be customized and added in "retiarii_nas/e2e_launch.py" and "retiarii_perf/artifact_start.py".

A.8 Notes

NVIDIA CUDA MPS may fail if a job is not stopped properly, which requires NVIDIA CUDA MPS to be restarted. Experiments in "retiarii_nas" will kill a running job for saving time, but may trigger the failure of NVIDIA CUDA MPS. We suggest to disable NVIDIA CUDA MPS when running experiments in "retiarii_nas".





KungFu: Making Training in Distributed Machine Learning Adaptive

Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, Peter Pietzuch
Imperial College London

Abstract

When using distributed machine learning (ML) systems to train models on a cluster of worker machines, users must configure a large number of parameters: hyper-parameters (e.g. the batch size and the learning rate) affect model convergence; system parameters (e.g. the number of workers and their communication topology) impact training performance. In current systems, adapting such parameters during training is ill-supported. Users must set system parameters at deployment time, and provide fixed adaptation schedules for hyper-parameters in the training program.

We describe *KungFu*, a distributed ML library for TensorFlow that is designed to enable adaptive training. KungFu allows users to express high-level *Adaptation Policies* (APs) that describe how to change hyper- and system parameters during training. APs take real-time monitored metrics (e.g. signal-to-noise ratios and noise scale) as input and trigger control actions (e.g. cluster rescaling or synchronisation strategy updates). For execution, APs are translated into monitoring and control operators, which are embedded in the dataflow graph. APs exploit an efficient asynchronous collective communication layer, which ensures concurrency and consistency of monitoring and adaptation operations.

1 Introduction

The popularity of machine learning (ML) in many application domains [3,15,37,75] has led to a wide adoption of distributed ML systems. Systems such as TensorFlow [1], PyTorch [60], MXNet [10] and MindSpore [53] exploit data and model parallelism [1,54,66,73] to train large ML models on clusters of worker machines. Training is typically done using the stochastic gradient descent (SGD) algorithm [43,68], which iteratively computes gradients to refine the model after each mini-batch of training data. ML systems compile training programs into dataflow graphs [1,10,30,60], which can be executed efficiently on GPUs and other accelerators.

When training ML models, users face the challenge of how to set a large number of *configuration parameters*, which split into two classes: *hyper-parameters* configure the training algorithm, such as SGD, and include the batch size [68], learning rate [68], momentum [63] and floating point precision [24]. Since hyper-parameters relate to the training process itself, their value affects the convergence rate and the final accuracy of the trained model. In addition, *system parameters* [42,73,81] control the operation of the distributed ML system, such as the number of workers, the partitioning of the training data, and the communication topology between workers. They impact the training performance, i.e. the time taken for the model to reach a given target accuracy.

Today users spend a substantial fraction of time tuning configuration parameters. Different ML models have different structures, and thus require different hyper-parameter settings [52]: the hyper-parameters for a vision model such as ResNet [26] differ from those for a language model such as BERT [15]. For each model, hyper-parameters such as batch size, learning rate and weight decay must be adjusted separately to reach a high model accuracy [52]. Approaches for automatic hyper-parameter tuning [2,18,35,45,52] search for the best settings offline at a high resource cost. Furthermore, system parameters such as the number of workers affect the resources consumed by training and their efficiency. Especially in a cloud setting, users must control resource usage to bound costs, while achieving good training performance [52].

Recently, we have seen a growing number of proposals [5,12,16,48,71] that argue for parameters to be adapted *dynamically* during training. For example, many models only reach high accuracy if the learning rate is decreased as the model converges [26,76]; the batch size can be set dynamically based on real-time gradient metrics [14,52,83]; and the communication strategy between workers can be adapted to the current training loss [72]. Similarly, system parameters can be updated to react to changes in exploitable levels of parallelism and resource availability. For example, the number of workers can be changed according to the observed resource utilisation, thus improving the utilisation of expensive accelerators such as GPUs and TPUs [46]; and the best communication topology among workers can be decided based on the available network bandwidth [56].

We observe that existing distributed ML systems and associated libraries (e.g. TensorFlow's Distribution Strategies [1], Horovod [73] and BytePS [8]) make it difficult to support the dynamic adaptation of configuration parameters for a number of reasons: (i) systems do not provide built-in mechanisms for adaptation. Users must rely on external frameworks, e.g. AutoScaling [58] adapts the number of machines by deploying extra scaling agents on each worker. Such external mechanisms are specialised to support only one type of adaptation. Since they are not integrated with the training system, they cannot take advantage of existing functionality and optimisations. In addition, (ii) the monitoring of training metrics introduces high overheads. For example, an 8-GPU server training a ResNet model produces 4 GB of gradients per second [55]. Any monitoring system (e.g. MLFlow [84]) that computes statistical metrics (e.g. variance [78]) over this amount of data consumes substantial compute resources and network bandwidth, which impacts the performance of the training process itself. Finally, (iii) the management of worker state with adaptation is challenging. In existing systems, users typically must checkpoint and restore all state when changing configuration parameters, which can take hundreds of seconds [58].

We describe *KungFu*,[1] a distributed ML training library that is designed to adapt configuration parameters at runtime. The key idea behind KungFu is to support *Adaptation Policies* (APs) written by users, which change hyper- and system parameters during training based on real-time monitored metrics. KungFu achieves this by making three contributions:

(1) Expressing Adaptation Policies. APs describe how configuration parameters should evolve based on monitored metrics. They are based on a high-level programming abstraction following the convention of existing ML frameworks, making integration with training environments seamless.

APs are written using *monitoring*, *communication* and *adaptation* functions: (i) monitoring functions compute metrics for gradients, model variables and worker performance; (ii) communication functions combine locally monitored metrics and transfer training state while adapting parameters; and (iii) adaptation functions update configuration parameters, including hyper- and system parameters.

(2) Making training monitoring efficient. KungFu supports the efficient monitoring of the training process, as needed by APs. Monitoring function calls are translated to *monitoring operators*, which are embedded in the execution dataflow graph. This allows monitoring operators to observe local gradients and reuse existing computation for monitoring.

Locally monitored gradients are combined to compute globally-aggregated metrics. This is achieved by an *asynchronous collective communication layer*, which avoids blocking the dataflow during monitoring. This layer uses a decentralised architecture: each worker maintains a local view of the state for collective communication and incrementally updates the state by exchanging messages in a peer-to-peer fashion. To maximise the performance of gradient monitoring, each KungFu worker has its own scheduler for collective communication. The schedulers cooperate in a decentralised fashion to exploit high-speed multi-GPU networks, e.g. as offered by NVLink through the NCCL interface.

(3) Distributed mechanism for adapting parameters. APs can adapt configuration parameters on distributed worker machines. KungFu represents configuration parameters as computational *configuration operators* embedded within the dataflow graph. In each training step, these operators can alter their output by reading configuration parameters provided by KungFu's asynchronous collective communication layer. Reading the parameters from this layer is efficient because it reuses existing data channels between the communication layer and the dataflow.

APs can dynamically change the parameters in the communication layer, and the result is automatically reflected in the dataflow. KungFu's communication layer uses a *distributed parameter adaptation algorithm* to protect the consistency of changes to configuration parameters while exploiting existing collective communication functions. These functions have been optimised for cross-machine communication, and thus allow adaptation to be performed with low latency.

We implement KungFu's communication layer and adaptation mechanisms in Go (~7k LOCs) and C++ (~3k LOCs), independently of the ML framework. KungFu provides Python bindings (~2k LOCs) for the Adaptation Policy interface, which can be used with existing ML frameworks, including TensorFlow [1], PyTorch [60] and Keras [11].

We evaluate experimentally the benefit and overhead of KungFu's Adaptation Policies. We show that KungFu users can implement a policy that dynamically adapts the batch size based on gradient noise scale, therefore significantly reducing the training time of a ResNet model. We also explore a policy that automatically searches for a cost-effective number of GPUs based on monitored worker performance when training a BERT model, reducing the cost by 20% compared to a static deployment. On a large-scale cloud testbed, we show that KungFu achieves negligible monitoring and adaptation overheads. It achieves up to 98% higher training throughput than Horovod, a state-of-the-art distributed ML system.

2 Adaptation in ML Systems

We first give background on distributed ML systems and their configuration parameters. We then describe current approaches for adapting parameters during training, highlighting why existing systems offer limited support for this.

2.1 Parameters in distributed ML systems

For many ML models, increasing the amount of training data and the size of the model improves accuracy [13, 26]. When training, ML systems therefore exploit the parallelism of modern hardware accelerators such as GPUs. Computation is typ-

[1] https://github.com/lsds/KungFu

ically expressed as a *dataflow* graph [1], which consists of individual operators that can be scaled out.

A supervised ML system trains a model using labelled samples, split into training and test data. A model gradually "learns" to predict the labels by adjusting its *model weights* based on the error. It takes several passes (or *epochs*) over the training data to minimise the prediction error. The test data is used to measure the model accuracy on previously unseen data. A key metric is *test accuracy*, which quantifies the model's ability to make predictions "in the wild".

The model weights are refined iteratively until the model achieves a desired test accuracy. Let w be a vector of the weights, and $\ell_{\mathsf{x}}(\mathsf{w})$ be a loss function that, given w, measures the difference between the predicted label of a sample (x,y) and the ground truth y. During training, an ML system tries to find a w^* that minimises the average loss e.g. using *mini-batch stochastic gradient descent* (SGD) [6, 7, 68]. More formally,

$$\mathsf{w}_{n+1} = \mathsf{w}_n - \frac{\gamma_n}{b} \sum_{\mathsf{x} \in \mathsf{B}_n} \nabla \ell_{\mathsf{x}}(\mathsf{w}_n) \tag{1}$$

where γ_n is the learning rate in the n-th iteration of the algorithm, B_n is a batch of b training samples, and $\nabla\ell$ is the gradient of the loss function, averaged over the batch samples.

To scale out the training computation across multiple CPUs or accelerators, ML systems can exploit data parallelism. In *parallel synchronous SGD* (S-SGD), K parallel workers share model replicas and compute gradients for distinct partitions of training data locally. Local gradients are averaged to correct the shared model:

$$\mathsf{w}_{n+1} = \mathsf{w}_n - \frac{\gamma_n}{Kb} \sum_{\mathsf{j}<\mathsf{K}} \sum_{\mathsf{x} \in \mathsf{B}_{n,j}} \nabla \ell_{\mathsf{x}}(\mathsf{w}_n) \tag{2}$$

The averaging of local gradients is usually implemented using *all-reduce* operations provided by collective communication libraries such as Horovod [73] and BytePS [8].

In a distributed ML system, the above training process is affected by many configuration parameters. These parameters can be placed into two groups: (i) accuracy-oriented *hyper-parameters* such as the learning rate γ_n, the batch size $|\mathsf{B}_n|$, momentum [63] and weight decay [38]; and (ii) performance-oriented *system parameters* such as the set of workers, their communication topology for performing synchronisation [56, 72, 73] and their roles during synchronisation, e.g. acting as primary and back-up workers to mitigate stragglers [9].

Hyper-parameters are properties that govern the training process and thus determine its final accuracy. They include variables that determine the model structure and how the network is trained (e.g. the learning rate). Choosing appropriate hyper-parameters plays a key role in training. For example, if the batch size is too high, the model may quickly settle at a local minimum and thus exhibit poor generalisation ability; conversely, if it is too low, the model may suffer from the noise of small batches and thus fail to converge.

System parameters affect the training throughput and thus the time to reach a given target accuracy. They include the configuration of workers (e.g. the number of parallel workers) and how they synchronise (e.g. the communication topology). Choosing appropriate system parameters is important for performance. For example, if the number of workers is too large, the system may suffer from low GPU utilisation due to communication bottlenecks; if the number of workers is too low, the training time for large models becomes prohibitively long.

2.2 Setting parameters in ML systems

Today users spend a substantial fraction of time setting configuration parameters [40]. They often search a large parameter space and decide on configuration parameters following a trial-and-error approach [26, 76]. Specifically, they empirically decide on a set of candidate values, and launch parallel training jobs to evaluate them [40]. They then measure the accuracy of the trained model and the system throughput, and eliminate under-performing settings using early-stop [40] and searching heuristics [29, 33, 41]. After that, they empirically choose an effective setting that reaches the target accuracy given a deadline or a resource budget.

When choosing candidate values for hyper-parameters, users must consider the characteristics of the datasets and models. For example, if the dataset is large (e.g. ImageNet [69]), the candidate batch size can be set larger (e.g. 2048) to improve the robustness of estimated gradients. If the dataset is small (e.g. CIFAR-10 [36]), the candidate batch size must be small (e.g. 64) so that it results in sufficiently many gradients to correct the model. Many large ML models (e.g. ResNet [26] and BERT [15]) have a complex loss space. When training such models, users often need to use a schedule of hyper-parameters (e.g. changing the learning rate at epochs 30, 60 and 90 for ResNet) to improve the quality of a found minima.

When choosing candidate system parameters, users consider the specification of hardware and the conditions of the network. For example, using a ring-based all-reduce topology among workers exploits the full host network bandwidth but it increases the depth of the topology, which adds to latency [80]. Setting the topology to be a star reduces latency but it requires larger bandwidth at the root node. In addition, good system parameters achieve a balance between compute and network utilisation. For example, in a cloud environment in which bandwidth is limited, training with NVIDIA V100 GPUs should only use few workers to prevent underutilising the expensive GPUs due to network bottlenecks; however, if NVIDIA K80 GPUs are used, a user would typically choose more workers to improve system throughput.

2.3 Dynamic adaptation of parameters

Recently, there has been a growing number of proposals to set configuration parameters *dynamically* based on metrics of the training process [5, 12, 16, 48, 71]. The idea is to incorporate

Class	Practitioners	Monitored metrics	Adaptation action
Accuracy	OpenAI [32, 52], Google [77]	Gradient noise scale	Scale **batch size** when the noise scale increases
	Kuaishou [47]	Gradient signal-to-noise ratio	Create online metrics for **model generalisation ability**
	Apple [31]	Gradient variance	Adapt **learning rate** based on the gradient variance
	DeepMind [4], NVIDIA [59]	Gradient second-order metrics	Adapt **learning rate** based on the second-order metrics
Performance	Microsoft [80], Uber [73]	Worker communication rate	Adapt **worker communication topology** based on rates
	Google [58], Huawei [82]	Worker utilisation	Adapt the **number of workers** based on utilisation
	Google [9], MIT [46]	Worker processing speed	Adapt the **roles of workers** based on straggler detection

Tab. 1: Recent proposals for the dynamic adaptation of parameters

knowledge about the training process and its progress through *gradient properties* and *performance metrics* on-the-fly.

Gradient properties include gradient signals, noise or derived signal-to-noise ratios, and they reflect the status of the trained model and the characteristics of the local loss space. They can be used to improve the setting of hyper-parameters, such as batch size and learning rates. Worker performance metrics, such as worker communication rate and processing rate, reflect the hardware and network conditions. They can be used to decide on the number of workers and their communication topology. Using monitored metrics to choose configuration parameters can significantly reduce the need for trial-and-error approaches when searching for suitable hyper-parameters. Instead of spending resources on a search process offline, fewer resources are used for the continuous calibration of configuration parameters during the learning process.

As summarised in Tab. 1, multiple proposals focus on adapting hyper-parameters to improve model accuracy. They often adapt critical hyper-parameters such as batch size and learning rate based on gradient properties. Researchers from OpenAI and Google Brain propose to monitor *gradient noise scale* to predict the optimal batch size when training deep learning models [32, 52, 77]; researchers at Kuaishou use the *gradient signal-to-noise ratio* to evaluate the generalisation ability of a model [47]; Apple automatically scales the learning rate based on gradient variance [31]; and DeepMind and NVIDIA use approximated metrics for *second-order gradients* to predict the best learning rate [4, 59]. Using such properties to set hyper-parameters has become important when training increasingly complex ML models. Users know little of the pre-conditions of these models, and hyper-parameters must be therefore set based on monitored properties [6].

Other proposals adapt system parameters to achieve higher training performance, e.g. reacting to changes in the exploitable parallelism and resource availability. As shown in Tab. 1, Microsoft and Uber propose to measure workers' *communication rates*, which are useful for optimising the topology of all-reduce operations [73, 79, 80]; Google and Huawei monitor *worker utilisation* to update the number of workers for increased resource utilisation [58, 82]; and Google detects straggling workers by analysing the distribution of *worker processing rates* and adapts the roles of workers, e.g. using backup workers to replace stragglers [9]. Using worker performance metrics to tune system parameters is an increasingly common practice. Many distributed ML systems are being deployed in cloud [25] and heterogeneous environments [23]. In such environments, the hardware specifications and network conditions are hard to predict, and thus system parameters must be adapted in the actual environment at runtime.

2.4 Open challenges

Although promising, proposals to adapt parameters are hard to realise in current systems, such as TensorFlow [1] and PyTorch [60]. Practitioners report three main challenges:

(1) No built-in mechanisms for adaptation. Existing distributed training libraries such as Horovod [73] provide insufficient mechanisms for adaptation. Users must rely on external systems that provide custom monitoring and adaptation components, which must be integrated into training systems: AutoScaling [58] adapts the number of workers at runtime by deploying extra scaling agents on each worker using a custom TensorFlow version, which can be managed by the scaling agents; Horovod Elastic [46] requires users to modify their existing training programs so that they can be executed by a custom elastic training runner.

In general, such external systems are specialised to support only a single type of adaptation, usually elasticity. They are not general adaptive training platforms with support for flexible monitoring and different types of adaptation (e.g. related to the communication topology). The lack of unified adaptation abstractions prevents adaptive training from leveraging existing ML system mechanisms and optimisations.

(2) High monitoring overhead. The dynamic adaptation proposals from Tab. 1 require fine-grained monitored metrics as input, but monitoring is expensive: an 8-GPU server training a ResNet model produces 4 GB of gradients per second [55], and this is even larger for recent language models such as BERT [15]. Shipping such an amount of gradient data from workers to a monitoring system such as TensorBoard [1], MLFlow [84], and Prometheus [64] consumes substantial network bandwidth. In addition, there is the overhead of computing complex statistical metrics (e.g. variance [78] or signal-to-noise ratios [47]) from gradients. All of this may impact the performance of the training process itself.

(3) Expensive state management under change. Workers maintain complex state, including model variables, hyper-

parameters and system parameters. This state must be managed carefully under adaptation: changing the number of workers must be reflected correctly in all dependent hyper-parameters, such as the learning rate and the data partitioning; otherwise the training result is affected adversely. In existing systems, users typically must checkpoint and restore all state when changing system parameters. This prevents users from extensively applying adaptation during training, as restoring the state can take hundreds of seconds [58].

3 Adaptation Policies

In this section, we introduce *Adaptation Policies* (APs), as supported by KungFu, which adapt configuration parameters based on monitored metrics. We provide an overview of the features of APs and describe the programming abstraction given to users to develop custom APs.

3.1 Overview

Our goals for APs are as follows: (i) we want to provide an expressive policy programming abstraction. The abstraction should follow conventions of existing ML frameworks. Users can thus develop their own policies with low effort. Moreover, (ii) we want to make policies easy to integrate with existing ML frameworks. This will allow users to choose policies based on their training scenarios and combine multiple polices for more advanced adaptation.

APs provide functions to help users implement custom monitoring and adaptation logic. Policies use *monitoring functions* to compute real-time metrics for worker performance and gradients. Locally monitored metrics can be combined using *communication functions*, which cover collective (broadcast and all-reduce) and point-to-point (serve and request) operations. Based on the monitored metrics, policies invoke *adaptation functions* to update the hyper-parameters and system parameters of the systems.

ML frameworks such as TensorFlow [1], Keras [11] and MXNet [10] provide a high-level training abstraction. Users call a generalised training method, which automatically trains a model until certain conditions (e.g. epoch counts) have been met. We want APs to be ported easily between ML frameworks, and we base APs on a framework-independent adaptation API (see Tab. 2).

To integrate this API with a framework, we observe that frameworks often support *user-defined callbacks* (e.g. Hooks in TensorFlow), which are repeatedly called during training. Today these callbacks have limited use—they usually implement checks for finishing conditions and logging functionality. KungFu's adaptation API can be implemented with callbacks, thus facilitating the integration with existing ML frameworks.

3.2 Sample AP for batch size adaptation

Next we describe a sample AP for dynamically increasing the batch size of S-SGD training based on online *gradient noise scale* (GNS) [52, 77]. The increase in batch size is implemented by adding extra workers. This allows the policy to increase training throughput, thus reducing completion time.

```
1  ... # Import ML framework libraries
2  import kungfu as kf
3
4  class GNSPolicy(kf.BasePolicy):
5    # Create policy state
6    def __init__(self, gns_opt):
7      self.opt = gns_opt
8      self.prev_gns = None
9      self.sync = True
10
11   # Synchronise model variables under adaptation
12   def before_epoch(self):
13     if self.sync:
14       for v in self.opt.variables():
15         v = kf.broadcast(v, 0) # Synchronise state
16       self.sync = False
17
18   # Adapt the number of workers if the GNS is growing
19   def after_epoch(self):
20     avg_gns = kf.allreduce(self.opt.gns()) / kf.size()
21     if self.prev_gns is None:
22       self.prev_gns = avg_gns
23     elif avg_gns > self.prev_gns:
24       new_size = int(kf.size() * avg_gns / self.prev_gns)
25       if new_size != kf.size():
26         kf.resize(new_size) # Scale the system
27         self.sync = True
28         self.prev_gns = avg_gns
29
30 model, data = ... # Import a model and a dataset
31 opt = SGDOptimizer(...)
32 opt = kf.OptimizerWithGNS(opt) # Embed monitoring
33 estimator = Estimator(model, opt, ...) # Create a trainer
34 policy = GNSPolicy(opt) # Instantiate the policy
35 estimator.train(data, hooks=[kf.PolicyHook([policy], ...)])
```

Listing 1: Sample Adaptation Policy for GNS

As shown Listing 1, the `GNSPolicy` is defined by extending a `BasePolicy` class (line 4). The policy includes the `__init__` function (line 6), which defines variables that maintain the policy state, such as the previously observed GNS metrics and a flag indicating if workers must synchronise their state (lines 7–9). The policy further has user-defined functions that trigger the adaptation logic at different times in a training process. The `before_epoch` function (line 12) is called at the start of each training epoch. Newly joined workers do not have state that is consistent with existing workers. It is thus necessary to broadcast (line 15) the model state. The `after_epoch` function (line 19) computes the averaged GNS metric at the end of each epoch using an all-reduce operation (line 20). Based on its value, the number of workers is increased by the `resize` function (line 26). To enable GNS monitoring, a user wraps the original `SGDOptimizer` with `kf.OptimizerWithGNS` (lines 31–33), which embeds the GNS monitoring operators into the training dataflow. The `GNSPolicy` is then passed to `PolicyHook` (line 35) to schedule its execution during training.

3.3 Adaptation Policy interface

To define APs, users implement custom *policy functions*. These functions can make API calls for *communication*, *monitoring* and *adaptation*, which are called at different times during the training process. There are three groups of pol-

Class	Functions	Description
Communication	**broadcast**(tensor, rank) → Tensor	Broadcast a tensor from a worker to all other workers
	allgather(tensor) → [Tensor]	Gather tensors from all workers and distribute the combined tensor to them
	allreduce(tensor) → Tensor	Aggregate tensors from all workers and distributes result back to them
	keep(tensor, tag)	Keep a tagged tensor which can be requested by other workers
	request(rank, name, tag) → Tensor	Request a tagged tensor from a specified worker
Monitoring	**comm_rates**() → Tensor	Measure tensor communication rates with other workers
	gns(grads, avg_grads) → float	Compute the gradient noise scale
	...	Custom gradient monitoring operators
Adaptation	**rank**() → int	Get the worker rank
	size() → int	Get the number of workers
	set_tree(tree) → bool	Set the tree of collective communication. Return True if succeed
	resize(size, workers=None) → bool	Resize the cluster based on a worker list. Return True if succeed
	detached() → bool	Check if the worker is detached due to resizing

Tab. 2: KungFu APIs for Adaptation Policies

icy functions: (i) the `before/after_train` functions are called at the start and end of a training job, respectively; (ii) the `before/after_epoch` functions are called at the start and end of each training epoch, respectively; and (iii) the `before/after_step` functions are called at the start and end of each training step (i.e. iteration), respectively.

In these policy functions, users can call APIs for training communication, monitoring and adaptation:

Communication. Tab. 2 lists the communication functions in APs. ML frameworks typically use tensors as their basic data types for gradients and model variables. To work with such data, the communication functions take tensors as inputs. APs need to collect monitored metrics from all workers, which can be achieved by calling *collective communication* functions: (i) the `broadcast` function distributes a tensor from a worker to all other workers; (ii) the `allgather` function gathers tensors from all workers and sends the combined tensor to all workers; and (iii) the `allreduce` function aggregates tensors from all workers and returns the results back to them.

In addition, APs must manage and communicate the state of trained models among workers. For example, APs for communication-efficient asynchronous training [44] or robust model averaging [85] must explicitly manage the lifecycle of model states and communicate states to synchronise diverged workers. To support state management and communication, the KungFu API provides a `keep` function that tags a model that is being trained (i.e. the state) and caches it in memory. APs can then read tagged models on other workers asynchronously using a `request` method.

Monitoring. Tab. 2 lists the monitoring functions, which APs use to monitor worker performance and gradients. APs use a `comm_rate` function to measure the tensor communication rates between a local worker and its peers. These rates are useful for deciding the optimal communication topology among workers. To monitor gradients, APs can use `gns` to compute the gradient noise scale. For other statistical metrics, such as variance, policies can use the above collective communication operators. For example, the computation of gradient variance requires both the sum of gradients and the sum of the square of gradients element-wise [78]; both summations can be computed using the `allreduce` function.

Adaptation. Based on monitored metrics, APs call adaptation functions to update configuration parameters. To update hyper-parameters, APs use the `allreduce` function to compute new values and assign them to hyper-parameters, represented as `params`. To update system parameters, APs call: (i) `set_tree` to set the collective communication topology among workers; and (ii) `resize` to update the number of workers. Some workers may need to leave the training after adaptation. APs can use the `detached` function to check if a local worker is still part of the training. If not, the AP can direct workers to exit gracefully.

3.4 Practical considerations

To support APs in real-world distributed ML systems, we must address several practical considerations:

Imperative and symbolic execution. To balance ease-of-use and performance, TensorFlow and PyTorch support imperative (TensorFlow Eager) and symbolic (TensorFlow AutoGraph and PyTorch TorchScript) execution, and APs must therefore also support both.

In TensorFlow Eager and PyTorch programs, users often want to customise the training process. Therefore they explicitly implement the training loop and call custom training functions (e.g. to compute gradients) imperatively in each iteration (i.e. step). This offers great flexibility but prevents callbacks from being used. In this case, KungFu allows the communication, monitoring and adaptation APIs from Tab. 2 to be called directly from inside the training loop.

To support symbolic execution, each function in Tab. 2 has a symbolic version. For example, the `resize` function has an equivalent symbolic version: `resize_op`. This allows KungFu APIs to be embedded into symbolic training programs (e.g. `tf.function`).

Listing 1 shows the hybrid usage of the KungFu imperative and symbolic APIs. The `OptimizerWithGNS` optimiser (line 32) appends `gns_op` operators to each gradient computation operator at compilation time of the dataflow graph, ensuring that the monitoring operator can execute immediately as long as its upstream gradient is available. The policy functions (lines 12–19) are called imperatively. This hybrid usage has an important advantage: the compute-intensive monitoring operators are embedded into the training dataflow graph, while the inexpensive user-defined adaptation logic can be triggered in different policy functions without re-compiling the dataflow graph.

Policy composition. Users can compose multiple APs to create advanced adaptive training applications (i.e. the `PolicyHook` (line 35 in Listing 1) can take multiple APs as input). For example, they can use two APs, one implementing elastic training (denoted as AP1) and the other an adaptive learning rate (denoted as AP2). These APs are chained as a list, which is passed to the training program. A current limitation is that users must decide manually on the correct invocation order: assuming AP1 modifies the worker count and AP2 uses the count to scale the learning rate, the execution order must be AP1 followed by AP2. We leave a mechanism for automatically determining the AP order to future work.

API restrictions. KungFu only imposes minimal API restrictions on AP developers, and APs can call any of the communication/monitoring/adaptation APIs in their callback functions. The calls have global atomic semantics, and there are no constraints on the call order. The only exception is that if a worker has left the cluster (checked by `detached`), it cannot further invoke collective communication APIs.

Error handling. AP developers must handle errors such as worker failures in a traditional fashion. If a KungFu API triggers an internal error, the exception is exposed as a dataflow error, as defined in existing ML frameworks, and checkpoints can be used for recovery.

4 Supporting Monitoring in KungFu

We describe *KungFu*, a distributed training library that can efficiently execute the proposed APs. APs must continuously monitor gradients to determine online adaption decisions, which must be done with low overhead. We begin with an overview of KungFu's design and then describe its support for efficient gradient monitoring in detail.

4.1 Design overview

To support monitoring, KungFu's design has the following goals: (i) KungFu should minimise extra computation when monitoring gradients, and worker resources should focus on training the model; (ii) KungFu should not block the training when monitoring gradients using collective communication operations due to the length of such operations; and (iii) KungFu should efficiently monitor gradients, given the large volume of gradients in today's models.

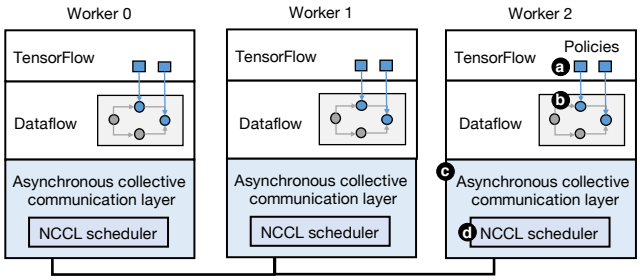


Fig. 1: KungFu architecture

Fig. 1 gives an overview of the KungFu architecture. Users can declare an AP (see ⓐ) as part of a ML training program written in TensorFlow. TensorFlow then creates a *dataflow* program to train the model. To monitor gradients in this dataflow, KungFu transforms the monitoring calls from the AP into *monitoring operators* (see ⓑ), which are embedded in the dataflow. This allows monitoring operators to (i) directly monitor gradients produced by the dataflow and (ii) reuse intermediate computation results in the dataflow for monitoring. For example, it becomes possible to exploit the existing averaged gradients computed when synchronising model replicas.

The monitoring process must compute globally-aggregated metrics from local gradients on workers. In KungFu, this exploits regular collective communication primitives (e.g. all-reduce and all-gather). To overlap monitoring and synchronisation as much as possible, KungFu has a new *asynchronous collective communication layer* (see ⓒ). Using this layer, the dataflow executed by workers can launch asynchronous collective communication operations without blocking.

The asynchronous collective communication layer also avoids having an expensive central coordinator, as used for invoking synchronous collective communication operations in existing systems, such as Horovod [73]. Instead, the KungFu communication layer follows a decentralised architecture: each worker maintains a local view of the complete cluster state used for collective communication and incrementally updates the state by exchanging messages with workers in a peer-to-peer fashion. This decentralised design avoids the need for APs to coordinate the order of collective communication across the system. It also prevents a central coordinator from becoming a potential bottleneck.

To improve the performance of collective communication, each KungFu worker has an NCCL scheduler (see ⓓ). This allows the worker to exploit high-speed multi-GPU networks, such as NVLink [57] and GPU RDMA, through the NCCL interface [56]. The scheduler tracks the availability of gradients on each GPU on the machine, and invokes a local NCCL library to execute a collective communication operation for fetching gradients. To combine the results on multiple workers across different machines, workers use KungFu's asynchronous collective communication layer, thus following a hybrid architecture for collective communication.

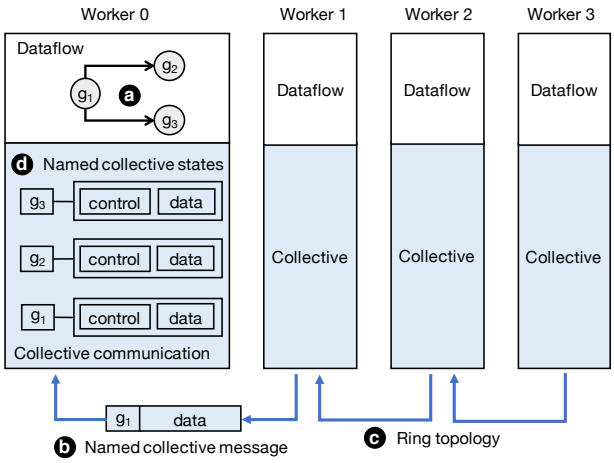


Fig. 2: Dataflow collective communication

4.2 Embedding monitoring within dataflows

To reduce the compute cost of calculating monitored metrics, KungFu exploits the fact that modern ML frameworks (e.g. TensorFlow, MXNet and PyTorch) have built-in *dataflow engines*. These engines offer efficient operators for tensor computation. They also handle the device placement of operators, leveraging parallel computation on accelerators such as GPUs and TPUs. Our observation is that a dataflow engine can also execute monitoring operators by embedding them within the dataflow graph for efficient execution.

To realise this design, KungFu implements the gradient monitoring functions (e.g. `gns` in Tab. 2) and the collective communication functions (e.g. `allreduce`, `broadcast` and `allgather`) as dataflow operators. Since gradients are represented as tensors in the dataflow graph, KungFu's dataflow operators must accept tensors as input. The embedding occurs at compilation time of the dataflow. The monitoring operators are thus part of the dataflow, and they can be scheduled immediately by the dataflow engine when their inputs, i.e. gradients, become available.

To embed its functions as operators, KungFu provides distributed optimisers (e.g. `OptimizerWithGNS` in line 32 in Listing 1) to wrap the original gradient descent optimisers. The KungFu optimisers automatically embed monitoring operators into the training dataflows. These operators intercept gradient tensors produced in each training iteration and forward them to gradient computation operators. The results are maintained in the dataflow and can be read subsequently by the policy functions in APs (line 20).

4.3 Collective communication for dataflows

Dataflows that implement APs use collective communication when computing global gradient metrics. While some gradient metrics (e.g. GNS) can be fused with synchronisation operations, others (e.g. gradient variance) cannot and require extra rounds of collective communication. Asynchronous collective communication thus allows these to be overlapped with gradient synchronisation, reducing the overhead of gradient monitoring. In addition, since dataflows are often executed asynchronously, the coordination with synchronous collective communication, as in Horovod, increases latency, which asynchronous communication avoids.

Allowing dataflows to launch collective communication asynchronously, however, can result in inconsistent computation. For example, the dataflows executed on different workers can produce gradients in different orders. If a worker receives the collective communication messages belonging to different gradients, they may compute inconsistent results.

Fig. 2 illustrates this problem. The example considers 4 workers that perform collective communication. They execute the same dataflow graph (shown as a), which contains 3 operators for computing gradients, g_1, g_2 and g_3. On different workers, the operators g_2 and g_3 can complete in a different order. To avoid mixing the collective communication data for g_2 and g_3, Horovod [73] employs a centralised coordinator. The coordinator tracks which gradients are ready on workers and launches collective communication operators for these in the correct order. This, however, not only reduces concurrency in the collective communication layer but it also makes the central coordinator a scalability bottleneck.

KungFu adopts a decentralised architecture that efficiently and safely implements asynchronous collective communication. It comprises several components:

Named collective messages. The collective communication layer in KungFu uses *named collective messages* (see b in Fig. 2) to communicate data. The delivery of these messages follows the collective communication topology (e.g. the ring topology shown in c). Each named collective message carries (i) the data and (ii) a key, which is used to identify which gradient the data belongs to. The key is derived from the unique key assigned by the ML framework to each dataflow operator. If such as key is unavailable, users can explicitly set it through KungFu's collective communication API.

Named collective states. When receiving a named collective message, a KungFu worker uses it to update its local *named collective state* (see d). The worker extracts the key from the message and identifies the state entry with the intermediate collective communication. Each entry contains a data and a control part: the data part is the buffer with the intermediate collective communication result, e.g. *max*, *min* or *sum*, which has been accumulated so far; the control part records how many named collective messages have been processed and which worker is the next hop to deliver the local intermediate collective communication results. If the worker finds itself as the last hop in the collective communication topology, it returns the result to the dataflow.

KungFu minimises the memory footprint of the collective messages and states. Since KungFu targets synchronous data parallel training, all asynchronous all-reduce operations must have completed in one training iteration before start-

ing the next. This limits the number of concurrent collective messages and states in memory. To further reduce memory consumption, KungFu frees the states and messages when an asynchronous all-reduce operation has completed. If possible, KungFu reuses buffers from the ML framework (TensorFlow/PyTorch), and it uses a pool to recycle buffers.

4.4 Accelerating collective communication with NCCL

High-end deep learning servers have fast communication links between GPUs (e.g. NVLink, which is 10× faster than PCIe [57]) and fast network connectivity between servers (e.g. GPU RDMA using InfiniBand). To speed up gradient communication and monitoring, KungFu workers exploit these fast links for collective communication.

In practice, users often employ NVLink and GPU RDMA through the NCCL collective communication library [56]. NCCL provides a synchronous collective communication API, following an MPI model [21]. At any time, an NCCL client can only launch a single collective communication operation; otherwise multiple NCCL operations interfere on the NVLink. Existing NCCL-enabled systems (e.g. Horovod-NCCL [73]) therefore adopt a centralised master architecture to coordinate distributed workers when using NCCL operations with gradients. This design, however, is not compatible with KungFu because its collective communication layer has a decentralised architecture.

Instead, KungFu workers use decentralised NCCL schedulers. Each scheduler tracks which gradients are ready on which GPU. The schedulers guarantee that gradients are processed by each NCCL instance in the same order. In the first training step, all NCCL schedulers monitor the order of gradients produced by local dataflow computations. They gather all orders and determine which order is most frequent. The most-common order (named *gradient order*) is broadcast to all schedulers. The schedulers must strictly follow the gradient order when calling NCCL. This ensures that NCCL schedulers launch collective communication for gradients consistently, without a need for central coordination.

KungFu currently offloads all collective communication requests, including those for gradient synchronisation and monitoring, to its NCCL schedulers if NVLink and InfiniBand are available locally. A future extension is to decide *which* requests to offload based on latency requirements: gradient monitoring could use asynchronous collective communication to overlap with training as much as possible; and throughput-intensive gradient synchronisation could use the NCCL-based collective communication.

5 Adapting Parameters of Workers

In this section, we describe how KungFu uses APs to adapt the parameters of its distributed workers.

5.1 Adapting dataflow parameters

Changing configuration parameters of a distributed ML system introduces challenges. Most systems require static parameters, which can be treated as constants when compiling the dataflow graph. After compilation, the dataflow graph is finalised and offloaded to GPUs for execution. Further changes to parameters are thus no longer reflected in the dataflow.

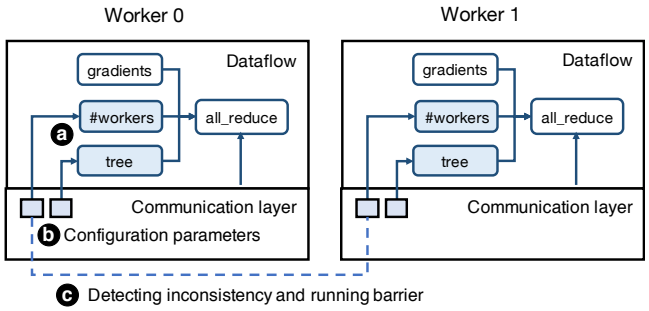


Fig. 3: Parameters as configuration operators

Therefore, elastic ML systems, such as Horovod Elastic [73] or Auto-Scaling [58], require users to use a *dynamic execution mode* of the ML framework, e.g. the "eager" mode in TensorFlow. The dynamic mode allows parameters to be updated in each training step, but it prevents the dataflow from being compiled, which results in large performance overheads. In addition, elastic ML systems only support changes to certain parameters, such as the number of workers. Users must still develop ad-hoc approaches when changing other parameters, such as the communication topology.

KungFu's design supports the online adaptation of dataflow parameters, while allowing the dataflow graph to be compiled. The core idea is that, instead of providing configuration parameters as static parameters when compiling the dataflow, KungFu adds parameters as computational *configuration operators* as part of the dataflow graph. In each training step, these configuration operators can dynamically alter their output by reading configuration parameters provided by KungFu's communication layer. This is efficient because it reuses existing data channels between the communication layer and the GPU. APs can dynamically change the parameters in the communication layer, and the result is reflected within the dataflow graph during execution.

Fig. 3 illustrates this idea. We assume that the dataflow graph is used to average local gradients, and it computes the sum of local gradients using an all-reduce operator. The AP changes (i) the number of workers and (ii) their collective communication topology. These two parameters are therefore provided as dataflow configuration operators (see ⓐ) and are used as the input to the all-reduce operator. During execution, the operators read the corresponding configuration parameters (see ⓑ) from the communication layer, and forward them to the all-reduce operator.

5.2 Protecting consistency under adaptation

APs must be able to change the configuration parameters in KungFu's distributed communication layer. At runtime, KungFu, however, must ensure that these parameters remain

Algorithm 1 Distributed adaptation algorithm for parameters

```
1:  procedure DistributedAdaptation(p, v)
2:      b ← bytes(v)                          ▷ Convert v into byte array
3:      l ← length(b)                         ▷ Get length of b
4:      l_0 ← allreduce(bytes(l), min)        ▷ byte-wise min
5:      l_1 ← allreduce(bytes(l), max)        ▷ byte-wise max
6:      if l_0 ≠ l_1 then                     ▷ byte-wise comparison
7:          return false
8:      end if
9:      b_0 ← allreduce(b, min)
10:     b_1 ← allreduce(b, max)
11:     if b_0 ≠ b_1 then
12:         return false
13:     end if
14:     p.update(v)                           ▷ Call the update function of p
15:     _ ← allreduce(['0'], min)             ▷ Run global barrier
16:     return true
17: end procedure
```

consistent when read by the distributed dataflows on workers.

Making global parameter changes consistent with APs introduces two requirements: (i) APs are replicated by workers and executed in parallel. They hold local monitoring state and can receive adaptation commands asynchronously. APs can thus obtain inconsistent values for a given parameter, especially in a large cluster in which many GPU workers asynchronously read new parameter values with high frequency. KungFu must have a mechanism to reject such inconsistent reads. In addition, (ii) when a consistent value is given, KungFu workers assign this value to their local parameters in parallel. The workers must then share a *global barrier* when completing the assignment, which prevents the execution of different dataflows with inconsistent values.

Distributed adaptation algorithm. We describe a distributed parameter adaptation algorithm that fulfils these requirements. To execute with low latency, thus reducing the time during which dataflow execution is blocked under adaptation, it exploits the collective communication layer: since configuration parameters are already hosted by that layer, KungFu re-uses the highly optimised collective communication functions to (i) detect inconsistent updates and (ii) implement a global barrier (shown as ❻ in Fig. 3).

Alg. 1 is executed by each KungFu worker when adapting a configuration parameter p with a new value v. It first transforms v into a byte array so that it can be consumed by an all-reduce function, together with a reduce function such as *min* or *max*. After that, the algorithm launches two all-reduce functions to check if the length of b is identical on all workers (lines 4–7). If so, it calls another two all-reduce functions to check if the content of b is consistent (lines 9–13). If this check also passes, v can be safely used for updating p (line 14). All workers must wait on a global barrier until the updates have completed. The barrier is implemented by calling an all-reduce function with a one-byte array (line 15).

Some parameters require custom adaptation logic other than a simple value assignment. For example, changing the number of workers requires workers to exit or join during adaptation. To support this, Alg. 1 can invoke a custom function when updating a parameter (line 14). In the case of the worker set, the function chooses one worker to signal other workers to exit or launch.

Managing data under adaptation. APs can modify the worker count and the batch size. These parameters affect how the training dataset is read and thus the training result. To ensure consistent results under adaptation, all KungFu workers have access to the full dataset.

KungFu supports two approaches to read data batches, depending on if users require *data epochs* to control the training process: (i) if data epochs are not needed, users can use random sampling to read data batches, and the adaptation logic can be triggered at any training step; (ii) with data epochs, KungFu provides a *dynamic data partitioning operator* that replaces the static partitioning operator (e.g. `tf.data.shard`) in the data input pipeline (e.g. DataSet). The dynamic partitioning operator is replicated on all KungFu workers and the operators are synchronised to enact a new parallelism level after a scaling operation. To preserve data epochs, users must invoke the adaptation logic on epoch boundaries only.

Handling failures during adaptation. To tolerate failures, KungFu relies on a highly-available configuration provider (e.g. `ConfigMap` in Kubernetes) to maintain its cluster configuration. The configuration must be updated when a scaling action is committed. In the case of worker failures, the cluster scheduler uses the configuration to restart workers.

6 Evaluation

We experimentally explore the following aspects of the KungFu design and implementation: (i) What are the benefits of enabling adaptation in distributed ML training? (ii) What is the monitoring and adaptation overhead in the training process? (iii) How does KungFu perform in large clusters compared to existing distributed ML systems?

6.1 Experimental set-up

We use both dedicated machines and cloud VMs in our experiments: the dedicated machines are (i) an NVIDIA DGX-1 machine with 8 NVIDIA V100 GPUs interconnected using NVLink, and 72 CPU cores; and (ii) a 20-CPU-core server with 4 NVIDIA Titan X GPUs interconnected using the PCIe bus. The cloud test-bed has 32 VMs, each with 8 vCPUs, 64 GB of memory and 1 NVIDIA K80 GPU.

We use various training workloads as part of the official models provided by TensorFlow [1]: the MobileNetV2 [70] and ResNet-50 [26] models for the ImageNet image classification task [37]; and the BERT [15] model for a natural language processing task, SQuAD [67]. The MobileNetV2 model is 23 MB, ResNet-50 is 98 MB, and BERT is 1 GB in size. These model sizes cover a large spectrum that users observe in practice. We use TensorFlow v1.13.2 to train the models. When possible, we compare the performance to Horovod v0.16.1.

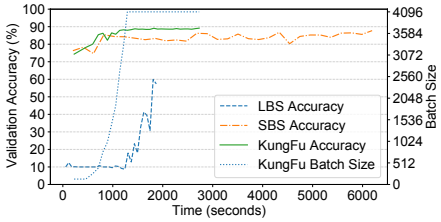


Fig. 4: Adaptive batch size (ResNet-56)

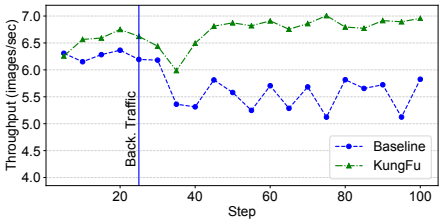


Fig. 5: Adaptive communication strategy (ResNet-50)

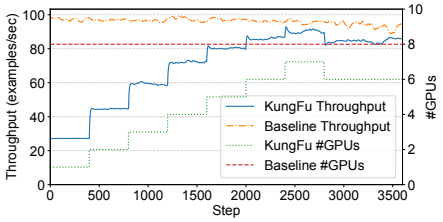


Fig. 6: Adaptive resource provisioning (BERT)

6.2 Adaptation policies

We evaluate three representative APs with KungFu that change various aspects of distributed training:

(1) Adaptive batch size. We implement an AP that adapts the batch size based on GNS when training the ResNet-56 model with the CIFAR-10 dataset. To the best of our knowledge, this AP is the first implementation that evaluates GNS-based batch size tuning in an online training scenario. Past work [52] only empirically evaluates it using offline training traces.

The AP computes GNS using an exponential moving average (α=0.1) and adapts the batch size every 10 epochs as follows: if GNS has increased by a factor of r, it also scales the batch size by r, up to 4096. We compare this AP with two static baselines, which adopt fixed batch sizes of 128 and 4096, respectively. These baselines represent typical choices for small batch size (SBS) and large batch size (LBS). In this experiment, the model is trained for 300 epochs with a learning rate of 0.1, based on TensorFlow's official model. The training is done on the 4 GPU Titan X testbed, with batches shared evenly across GPUs.

Fig. 4 shows the validation accuracy of the model over time. LBS reaches a low validation accuracy (60%) but finishes quickly. SBS reaches a higher validation accuracy (88%) but the constant noise in gradients due to the small batches makes it hard to converge, and the accuracy oscillates between 80% and 90%. A typical issue of SBS is the underutilisation of GPUs: SBS takes around 6000 s to complete training, 2.4× longer than LBS. In practice, fixing the choice of batch size is challenging for users—they have to trade off between model accuracy and hardware utilisation.

The above AP addresses this challenge. As shown by the right y-axis in Fig. 4, the policy dynamically increases the batch size from 128 to 4096 based on GNS. This type of adaptation improves model accuracy: it reaches 88% after around 1000 s, 5× faster than SBS, and eventually converges to 90% after 1300 s. Dynamically increasing the batch size reduces the noise in gradients, which enables the model to converge. Furthermore, it allows the model to better utilise the hardware: the model spends only 400 s more than LBS but achieves 30% higher accuracy.

(2) Adaptive communication strategy. Network infrastructure in cloud environments and multi-tenant clusters may suffer from contention when using all-reduce operations to synchronise gradients, and straggling workers may then slow down the entire system [49]. To address this, we provide an AP that monitors training throughput. If the throughput drops due to network contention, the policy adjusts the topology used by all-reduce, limiting the use of contended network links. In this experiment, we train the ResNet-50 model for 100 steps on 32 VMs. After 25 steps, we introduce background traffic to create network contention. This mimics a cloud environment in which there is dynamic interference in an over-subscribed network.

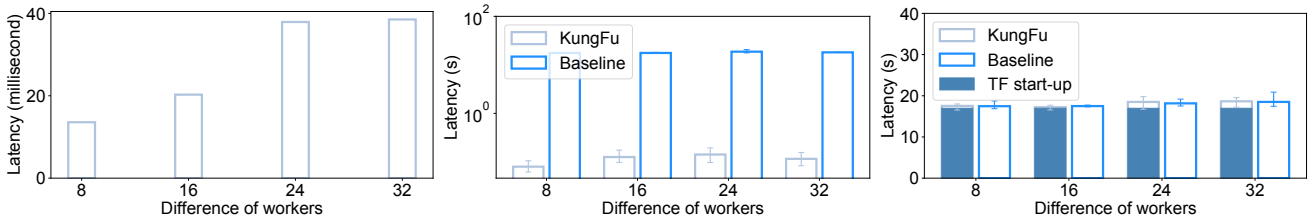
We compare this AP with a static baseline that uses a fixed all-reduce topology. We also attempted to implement a dynamic baseline using OpenMPI and NCCL, but these libraries do not allow runtime control of the all-reduce topology.

Fig. 5 presents the average worker training throughput over training steps. The baseline shows that the workers reach 6.5 images/s at the beginning but this number drops to 5.5 after the network becomes contended. The AP monitors the throughput and detects network contention at step 35. It adapts the communication topology, and the topology recovers throughput: it increases to 7 images/s, even though the background traffic is still on-going.

(3) Adaptive resource provisioning. Users want to decide on a cost-effective number of GPUs when training models. Using many GPUs leads to high training throughput but it also increases cost. Large ML models are synchronising large volumes of gradients. Above a certain amount of resources, communication becomes a bottleneck. In such a case, using more GPUs only gives a marginal performance improvement, despite the higher cost.

We explore an AP that finds the most cost-effective number of GPUs. This policy adds one worker every K steps. It then evaluates the average total accumulated throughput and, if the new throughput is not $1+\alpha(1/\text{size})$ times higher, it removes the worker and stops scaling. We choose α=0.33 and K=400. We compare to a static baseline that always uses the most GPUs. We train the BERT model with a per-GPU batch size of 8 on the 8 GPU V100 testbed.

Fig. 6 shows the results. When all 8 GPUs (right-hand y-axis) are used from the beginning, the total throughput is above 90 examples/second (left-hand y-axis). For KungFu, we see that the throughput rises with the number of GPUs until only a slight increase from 6 to 7 GPUs (step 2400). Due to the small increase with 7 workers, KungFu removes worker 7 at step 2800, stops scaling and resumes training



(a) Set tree latency (b) Scale in latency (c) Scale out latency

Fig. 7: Adaptation overhead

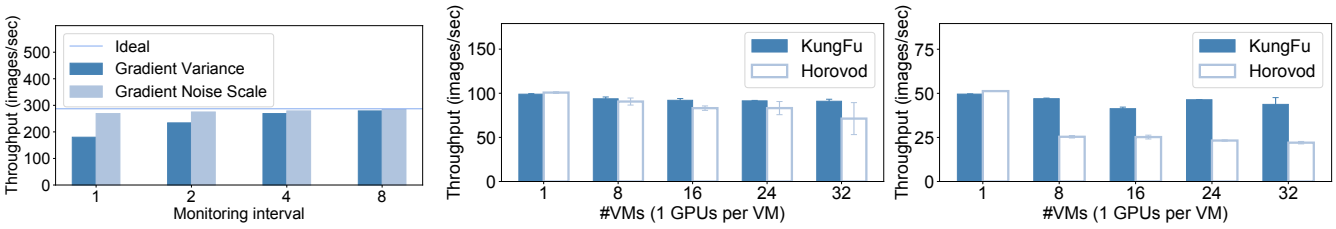


Fig. 8: Monitoring overhead

(a) MobileNetV2 (b) ResNet-50

Fig. 9: Worker throughput with different cluster sizes

with 6 workers. We use the price of a V100 GPU on Azure to estimate the cost efficiency of KungFu and the baseline. The baseline has a cost efficiency of 10,902 examples/USD, and KungFu has 13,097 examples/USD. This indicates that KungFu improves the cost efficiency of the job by 20%.

6.3 Adaptation overhead

Next we evaluate the overhead of adaptation and monitoring.

Adaptation. We evaluate the adaptation latency when changing the communication topology and worker set. We conduct the experiments on the 32 VMs testbed and train ResNet-50. During training, we repetitively change the parameters.

Fig. 7a shows the latency when changing worker communication topology. With 8 VMs, the adaptation completes in 15 ms. With 32 VMs, the delay only increases to 37 ms. This shows the benefits of using the all-reduce function to implement the required consistency checking and global barrier during adaptation.

Fig. 7b shows the latency when scaling down. We decrease the number of workers from 32 to 1 by calling the `resize` function. The function takes 0.2 s to complete. Scaling the system using the checkpoint/recovery mechanism of TensorFlow takes around 20 s to complete, 100× slower than KungFu. This high latency is mainly due to the stop-and-resume time of TensorFlow, and it is consistent with the observations made by others [58, 82]. This result shows the need for supporting efficient adaptation to enable scaling in practice.

Fig. 7c shows the latency when scaling out. Increasing the number of workers from 1 to 32 takes 20 s, the same as the baseline. Since KungFu must preserve the consistency of the training state on workers, it must wait for new workers to be started by TensorFlow. Breaking down this delay, we can see that KungFu spends 0.5 s to complete the scale-out operation, and waits the remaining 19.5 s for the TensorFlow set-up. The long start-up time of TensorFlow can be masked by implementing worker pre-loading [58].

Monitoring. We also consider the overhead when monitoring two metrics, *gradient noise scale* (GNS) and *gradient variance* (GV). The computation of GNS can reuse the averaged gradients produced by the S-SGD computation and thus can be computed locally without extra collective communication; GV, however, compares the square of the sum of gradients and the sum of gradient squares [78]. To compute it, KungFu must launch an additional all-reduce operation for each gradient. We monitor these two metrics when training ResNet-50 for ImageNet on the 8 GPUs V100 testbed. We vary the monitoring interval from 1–8 steps to change load.

Fig. 8 shows the average per-worker training throughput with gradient monitoring. We compare it to the per-worker throughput without monitoring (i.e. the ideal case). The monitoring of GNS has a negligible impact on training, dropping the training throughput from 6.3% to 1.0% based on the monitoring interval. This shows that embedding the monitoring operators as part of the dataflow graph results in low overhead.

The calculation of GV has a tangible throughput impact. The overhead, however, can be amortised by increasing the monitoring interval. The throughput drops by 2.8% when the interval is 8 steps, while still providing acceptable monitoring for APs. APs keep monitored metrics in data sketches and use the accumulated result, usually every several epochs. Iterating through an ImageNet dataset takes more than 40,000 steps, which means that 5000 GV values in an epoch still make estimation reliable.

6.4 Performance

Finally, we evaluate two aspects of KungFu that contribute to overall performance: (i) the asynchronous collective communication layer and (ii) the NCCL scheduler.

Asynchronous Collective Communication Layer. We explore how the performance of KungFu's communication layer compares to Horovod [73], which is a popular high-performance collective communication library used for distributed ML training. We compare the performance with 8, 16, 24 and 32 VMs. By varying the cluster size, we place different loads on collective communication.

Fig. 9a shows the per-VM training throughput for MobileNetV2/ImageNet under KungFu and Horovod. With 8 VMs, Horovod and KungFu achieve the same throughput. With 32 VMs, however, KungFu outperforms Horovod by 28% due to the benefits of its decentralised design for the communication layer, which avoids the bottleneck of Horovod's master. We also note that Horovod shows a high variance in the training throughput for 32 VMs (up to 24% between min/max). This is caused by network jitter in the cloud environment affecting Horovod's coordinator. Since KungFu workers asynchronously exchange messages for collective communication, they compensate for the network latency and thus achieve stable training throughput even with 32 VMs.

Fig. 9b shows the per-VM training throughput for ResNet50-ImageNet, which is 4× larger than MobileNetV2. With this model, there is more network traffic, and KungFu achieves 98% higher throughput than Horovod with 32 VMs. This improvement is larger than in the case of MobileNetV2 because Horovod must execute collective communication in order, following the MPI convention. KungFu, however, supports concurrent collective communication operations through its named collective message and state mechanisms. This increases concurrency in the communication layer, making it achieve a higher throughput than Horovod, especially with large models such as ResNet.

NCCL Schedulers. We also explore the benefit of the NCCL schedulers in comparison to CPU-based collective communication (i.e. CPU all-reduce) and centralised NCCL scheduling, as used by Horovod-NCCL. The experiment is executed on the DGX-1 machine with all 8 NVIDIA V100 GPUs.

Fig. 10 shows the throughput with CPU-based collective communication and NCCL as used by KungFu and Horovod, respectively. For communication between GPUs on the same machine, NCCL offers a significant performance benefit for both KungFu and Horovod. Comparing KungFu and Horovod with NCCL, we see that, for ResNet (~200 gradients; 97 MB size), KungFu and Horovod experience almost identical performance; for BERT-base (~600 gradients; 1 GB size), KungFu achieves 17% higher throughput than Horovod.

This difference can be attributed to the centralised nature of Horovod's NCCL scheduling. The central scheduler contacts each worker to track which gradients have become available. When gradients are available on all workers, the scheduler calls an all-reduce operation to average gradients. The scheduling overhead grows with the number of gradients, and it becomes a bottleneck with many gradients (e.g. 600 gradients in BERT). A large number of gradients is increasingly common because large models out-perform smaller ones.

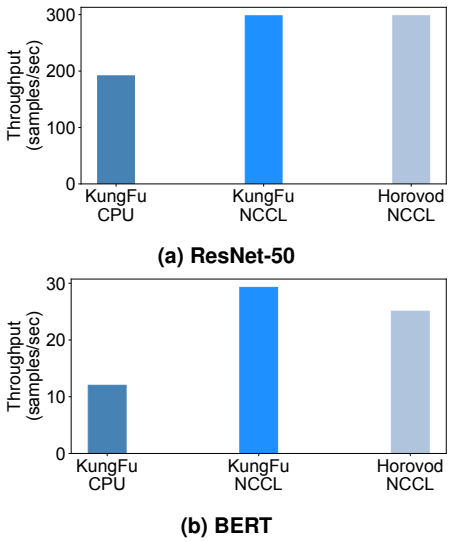


Fig. 10: Training throughput

7 Related Work

Distributed ML systems. A dataflow abstraction is used in many ML systems, including TensorFlow, PyTorch [60], MXNet [10], Caffe [30] and MindSpore [53]. These systems share similar dataflow designs in which computational operators are used for tensor computation. Compiled dataflows are offloaded to GPUs for the training computation. KungFu reuses the dataflow abstraction to embed operators for the purpose of adaptation.

KungFu uses collective communication functions to implement monitoring and adaptation operations. Such functions are available in most distributed ML systems, including those built on top of MPI [1, 56, 73] as well as parameter-server-based systems [8, 42]. Compared to existing collective communication approaches, KungFu explores a decentralised architecture that is tailored to supporting dataflows used in ML frameworks. It allows multiple collective communication operations to execute concurrently, making it different from current MPI-compatible systems.

Hyper-parameter optimisation and tuning. To find the best settings for hyper-parameters, practitioners and researchers have proposed tuning systems [2, 17, 18, 35, 40, 45] with associated search algorithms [28, 29, 33, 41]. These systems launch parallel training jobs to evaluate different candidate settings of target hyper-parameters. They often aim to minimise resource consumption for finding the best setting. In contrast, KungFu explores how to optimise hyper-parameters continuously in a single training job. It thus proposes mechanisms for efficient monitoring and online adaptation of hyper-parameters during training. It can be used by existing tuning systems to speed up the time of individual training jobs.

Elastic training systems have been proposed to improve the resource utilisation of ML clusters. EDL [82] studies stop-

free scaling for TensorFlow workers, and Litz [65] proposes an elastic training framework for ML clusters that consist of parameter servers and training workers. Horovod Elastic [73] and PyTorch Elastic [60] are two open-source elastic training libraries. Compared to these dedicated elastic training systems, KungFu provides a unified framework that can execute different adaptive training jobs efficiently.

Adaptation policies have been explored in streaming systems [20, 22, 27, 61, 62]. Dhalion [19] provides policy support for Apache Storm, and its policies measure data analytics metrics, such as latency and throughput; in contrast, APs in KungFu enable the monitoring of gradients in ML systems. Chi [51] is a control plane for stream processing systems, and it supports online monitoring and adaptation. Compared to Chi, KungFu provides a solution to build adaptive distributed ML systems with high-performance gradient monitoring using dataflows and asynchronous collective communication.

Recently, practitioners have proposed adaptation policies [39, 58] tailored to ML systems. These policies use cost models to infer the performance of a training system and make scaling decisions in response. They could be implemented as APs on top of KungFu to exploit its optimised adaptation and communication infrastructure.

Monitoring training. The ML communities have recognised the importance of monitoring training [71]. CrossBow [34] monitors accelerator utilisation to infer the optimal level of data parallelism when training models. Moreover, gradients metrics are useful to optimise hyper-parameters [50]. There have been efforts on setting the batch size according to signal-to-noise ratios [12] and loss [5], or the learning rate based on other gradient metrics, e.g. square norm of expectation of gradients [16, 71, 74]. Due to the lack of support in current distributed ML systems, such efforts typically only evaluate efficacy using offline collected gradients. KungFu is inspired by these efforts and addresses the missing systems support to implement such proposals.

8 Conclusions

When training modern complex ML models, users want to adapt a wide range of hyper- and system parameters. Existing distributed ML systems were designed at a time when static training regimes were the norm. They thus lack mechanisms for monitoring training metrics and adapting configuration parameters at runtime.

We have presented *KungFu*, a distributed training library that allows users to specify and execute Adaptation Policies. KungFu executes policies efficiently by embedding monitoring and configuration operators as part of the compiled dataflow graph. All communication leverages efficient asynchronous collective communication functions, without interfering with the training process or compromising consistency.

Acknowledgements. We thank our shepherd, Derek Murray, for his thoughtful and detailed comments on the paper.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, Savannah, GA, USA, 2-4 November 2016.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A Next-generation Hyperparameter Optimization Framework. In *25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 2623–2631, Anchorage, AK, USA, 4-8 August 2019.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, and Guoliang Chen. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. In *33rd International Conference on Machine Learning (ICML)*, volume 48 of *Proceedings of Machine Learning Research*, pages 173–182, New York, New York, USA, 20-22 June 2016.

[4] Jimmy Ba, Roger B. Grosse, and James Martens. Distributed Second-Order Optimization using Kronecker-Factored Approximations. In *5th International Conference on Learning Representations (ICLR)*, Toulon, France, 24-26 April 2017.

[5] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling Adaptive Batch Sizes with Learning Rates. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 410–419, 11-15 August 2017.

[6] L. Bottou, F. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 60(2):223–311, 2018.

[7] Léon Bottou. On-line Learning and Stochastic Approximations. In *On-line Learning in Neural Networks*, pages 9–42. New York, NY, USA, 1998.

[8] ByteDance. BytePS - A High Performance and Generic Framework for Distributed DNN Training. https://github.com/bytedance/byteps, 2020.

[9] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. Revisiting Distributed Synchronous SGD. *CoRR*, abs/1604.00981, 2016.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[11] François Chollet. Keras. https://keras.io, 2015.

[12] Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Automated Inference with Adaptive Batches. In *20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1504–1513, Fort Lauderdale, FL, USA, 20–22 Apr 2017.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, and Ke Yang. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, pages 1223–1231. 2012.

[14] Aditya Devarakonda, Maxim Naumov, and Michael Garland. AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks. *CoRR*, abs/1712.02029, 2017.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 4171–4186, Minneapolis, MN, USA, 2-7 June 2019.

[16] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. In *23rd Conference on Learning Theory (COLT)*, pages 257–269, Haifa, Israel, 27-29 June 2010.

[17] Raul Castro Fernandez, William Culhane, Pijika Watcharapichat, Matthias Weidlich, Victoria Lopez Morales, and Peter R. Pietzuch. Meta-Dataflows: Efficient Exploratory Dataflow Jobs. In *International Conference on Management of Data (SIGMOD)*, pages 1157–1172, Houston, TX, USA, 10-15 June 2018.

[18] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28*, pages 2962–2970. 2015.

[19] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-Regulating Stream Processing in Heron. *Proc. VLDB Endow.*, 10(12):1825–1836, August 2017.

[20] Tom Z J Fu, Jianbing Ding, Richard T B Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams. In *35th International Conference on Distributed Computing Systems (ICDCS)*, volume 2015-July, pages 411–420, Columbus, Ohio, USA, 29 June - 2 July 2015.

[21] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Lecture Notes in Computer Science*, volume 3241, pages 97–104, USA, 2004.

[22] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2014.

[23] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR*, abs/1706.02677, 2017.

[24] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *32nd International Conference on Machine Learning (ICML)*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1737–1746, Lille, France, 6-11 July 2015.

[25] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Addressing the Straggler Problem for Iterative Convergent Parallel ML. In *7th ACM Symposium on Cloud Computing (SoCC)*, pages 98–111, 5-7 October 2016.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, 27-30 June 2016.

[27] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 261–272, Asilomar, CA, USA, 9-12 January 2011.

[28] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol

Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. *CoRR*, abs/1711.09846, 2017.

[29] Kevin Jamieson and Ameet Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *19th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 51 of *Proceedings of Machine Learning Research*, pages 240–248, Cadiz, Spain, 9-11 May 2016.

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM International Conference on Multimedia (MM)*, pages 675–678, Orlando, FL, USA, 03-07 November 2014.

[31] Tyler B. Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. AdaScale SGD: A User-Friendly Algorithm for Distributed Training. *CoRR*, abs/2007.05105, 2020.

[32] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *CoRR*, abs/2001.08361, 2020.

[33] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost Optimal Exploration in Multi-Armed Bandits. In *30th International Conference on Machine Learning (ICML)*, volume 28 of *Proceedings of Machine Learning Research*, pages 1238–1246, Atlanta, Georgia, USA, 17-19 June 2013.

[34] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. Crossbow: Scaling Deep Learning With Small Batch Sizes on Multi-Gpu Servers. *Proceedings of the VLDB Endowment*, 12(11):1399–1412, 2019.

[35] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-WEKA 2.0: Automatic Model Selection and Hyperparameter Optimization In WEKA. *Journal of Machine Learning Research*, 18(25):1–5, 2017.

[36] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto, 2009.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances Neural Information Processing Systems 24*, pages 1097–1105, USA, 2012.

[38] Anders Krogh and John A. Hertz. A Simple Weight Decay Can Improve Generalization. In *Advances in Neural Information Processing Systems 4*, pages 950–957. 1992.

[39] Woo-Yeon Lee, Yunseong Lee, Joo Seong Jeong, Gyeong-In Yu, Joo Yeon Kim, Ho Jin Park, Beomyeol Jeon, Wonwook Song, Gunhee Kim, and Markus Weimer. Automating System Configuration of Distributed Machine Learning. In *39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2057–2067, Dallas, Texas, USA, 7-9 July 2019. IEEE.

[40] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A System for Massively Parallel Hyperparameter Tuning. In *Machine Learning and Systems (MLSys)*, pages 230–246. 2-4 March 2020.

[41] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.*, 18(1):6765–6816, January 2017.

[42] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning With the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 583–598, Broomfield, CO, USA, 6-8 October 2014.

[43] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient Mini-batch Training for Stochastic Optimization. In *20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (DMKD)*, pages 661–670, New York, NY, USA, 2014.

[44] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *35th International Conference on Machine Learning (ICML)*, volume 80, pages 3043–3052, Stockholm, Sweden, 10-15 July 2018.

[45] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training. *CoRR*, abs/1807.05118, 2018.

[46] Haibin Lin, Hang Zhang, Yifei Ma, Tong He, Zhi Zhang, Sheng Zha, and Mu Li. Dynamic Mini-batch SGD for Elastic Distributed Training: Learning in the Limbo of Resources. *CoRR*, abs/1904.12043, 2019.

[47] Jinlong Liu, Guoqing Jiang, Yunzhi Bai, Ting Chen, and Huayan Wang. Understanding Why Neural Networks Generalize Well Through GSNR of Parameters. *CoRR*, abs/2001.07384, 2020.

[48] Maren Mahsereci and Philipp Hennig. Probabilistic Line Searches for Stochastic Optimization. In *Advances in Neural Information Processing Systems 28*, pages 181–189. 2015.

[49] Luo Mai, Chuntao Hong, and Paolo Costa. Optimizing Network Performance in Distributed Machine Learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Santa Clara, CA, USA, 6-7 July 2015.

[50] Luo Mai, Alexandros Koliousis, Guo Li, Andrei-Octavian Brabete, and Peter R. Pietzuch. Taming Hyperparameters in Deep Learning Systems. *ACM SIGOPS Oper. Syst. Rev.*, 53(1):52–58, 2019.

[51] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, and Vamsi Kuppa. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proceedings of the VLDB Endowment*, 11(10):1303–1316, 2018.

[52] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An Empirical Model of Large-Batch Training. *CoRR*, abs/1812.06162, 2018.

[53] MindSpore. Mindspore Deep Learning Training/Inference Framework. https://github.com/mindspore-ai/mindspore, 2020.

[54] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism For DNN Training. In *27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Huntsville, ON, Canada, 27-30 October 2019.

[55] NVIDIA. Data Center Deep Learning Product Performance. https://developer.nvidia.com/deep-learning-performance-training-inference, 2020.

[56] NVIDIA. Optimized Primitives for Collective Multi-GPU Communication. https://github.com/NVIDIA/nccl, 2020.

[57] NVIDIA. The Building Blocks of Advanced Multi-GPU Communication. https://www.nvidia.com/en-us/data-center/nvlink/, 2020.

[58] Andrew Or, Haoyu Zhang, and Michael J. Freedman. Resource Elasticity in Distributed Deep Learning. In *Machine Learning and Systems (MLSys)*, Austin, TX, USA, 2-4 March 2020.

[59] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12359–12367, Long Beach, CA, USA, 16-20 June 2019.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, and Luca Antiga. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. 2019.

[61] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research (CIDR)*, Chaminade, CA, USA, 8-11 January 2017.

[62] Thao N. Pham, Panos K. Chrysanthis, and Alexandros Labrinidis. Avoiding Class Warfare: Managing Continuous Queries With Differentiated Classes of Service. *VLDB J.*, 25(2):197–221, 2016.

[63] Boris Polyak. Some Methods of Speeding up the Convergence of Iteration Methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.

[64] Prometheus. The Prometheus Monitoring System and Time Series Database. https://github.com/prometheus/prometheus, 2019.

[65] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic Framework for High-Performance Distributed Machine Learning. In *USENIX Annual Technical Conference (ATC)*, pages 631–644, Boston, MA, 11-13 July 2018.

[66] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *CoRR*, abs/1910.02054, 2019.

[67] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know What You Don't Know: Unanswerable Questions for

SQuAD. In *56th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 784–789, Melbourne, Australia, 15-20 July 2018.

[68] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.

[69] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[70] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, Salt Lake City, UT, USA, 18-22 June 2018.

[71] Tom Schaul, Sixin Zhang, and Yann LeCun. No More Pesky Learning Rates. In *30th International Conference on Machine Learning (ICML)*, volume 28 of *Proceedings of Machine Learning Research*, pages 343–351, Atlanta, Georgia, USA, 17-19 June 2013.

[72] Vetter Scott, Elpelt Tobias, Franke Rico, and Miranda Yanil Z. Networking Design for HPC and AI on IBM Power Systems (Red Paper), IBM PowerAI Distributed Deep Learning. http://www.redbooks.ibm.com/redpapers/pdfs/redp5478.pdf, April 2018.

[73] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *CoRR*, abs/1802.05799, 2018.

[74] Ravid Shwartz-Ziv and Naftali Tishby. Opening the Black Box of Deep Neural Networks via Information. *CoRR*, abs/1703.00810, 2017.

[75] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, and Marc Lanctot. Mastering the Game of Go With Deep Neural Networks and Tree Search. *Nature*, 529:484–503, 2016.

[76] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't Decay the Learning Rate, Increase the Batch Size. *CoRR*, abs/1711.00489, 2017.

[77] Samuel L. Smith and Quoc V. Le. A Bayesian Perspective on Generalization and Stochastic Gradient Descent. In *6th International Conference on Learning Representations (ICLR)*, Vancouver, BC, Canada, 30 April - 3 May 2018.

[78] Y. Tsuzuku, Hi. Imachi, and T. Akiba. Variance-based Gradient Compression for Efficient Distributed Deep Learning. In *6th International Conference on Learning Representations (ICLR)*, Vancouver, BC, Canada, 30 April - 3 May 2018.

[79] Marcel Wagenländer, Luo Mai, Guo Li, and Peter R. Pietzuch. Spotnik: Designing Distributed Machine Learning for Transient Cloud Resources. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 13-14 July 2020.

[80] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil R. Devanur, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. *CoRR*, abs/1910.04940, 2019.

[81] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised Deep Learning with Partial Gradient Exchange. In *7th ACM Symposium on Cloud Computing (SoCC)*, SoCC '16, pages 84–97, New York, NY, USA, 5-7 October 2016.

[82] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, and James Cheng. Elastic Deep Learning in Multi-Tenant GPU Cluster. *CoRR*, abs/1909.11985, 2019.

[83] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-Batch Training for LSTM and Beyond. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 9:1–9:16, Denver, Colorado, USA, 17-19 November 2019.

[84] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Engineering Bulletin*, 41(4):39–45, 2018.

[85] Michael R. Zhang, James Lucas, Jimmy Ba, and Geoffrey E. Hinton. Lookahead Optimizer: k Steps Forward, 1 Step Back. In *Advances in Neural Information Processing Systems 32*, pages 9593–9604, Vancouver, BC, Canada, 8-14 December 2019.

FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization

Dongup Kwon[1,2], Junehyuk Boo[1], Dongryeong Kim[1], Jangwoo Kim[1,2,*]

[1]*Department of Electrical and Computer Engineering, Seoul National University*
[2]*Memory Solutions Lab, Samsung Semiconductor Inc.*

Abstract

Emerging big-data workloads with massive I/O processing require fast, scalable, and flexible storage virtualization support. Hardware-assisted virtualization can achieve reasonable performance for fast storage devices, but it comes at the expense of limited functionalities in a virtualized environment (e.g., migration, replication, caching). To restore the VM features with minimal performance degradation, recent advances propose to implement a new software-based virtualization layer by dedicating computing cores to virtual device emulation. However, due to the dedication of expensive general-purpose cores and the nature of host-driven storage device management, the proposed schemes raise the critical performance and scalability issues with the increasing number and performance of storage devices per server.

In this paper, we propose FVM, a new hardware-assisted storage virtualization mechanism to achieve high performance and scalability while maintaining the flexibility to support various VM features. The key idea is to implement (1) a storage virtualization layer on an FPGA card (FVM-engine) decoupled from the host resources and (2) a device-control method to have the card directly manage the physical storage devices. In this way, a server equipped with FVM-engine can save the invaluable host-side resources (i.e., CPU, memory bandwidth) from virtual and physical device management and utilize the decoupled FPGA resources for virtual device emulation. Our FVM-engine prototype outperforms existing storage virtualization schemes while maintaining the same flexibility and programmability as software implementations.

1 Introduction

Storage virtualization is one of the most important components to determine the cost-effectiveness of modern datacenters, which improves the utilization of the storage devices and makes resource management much easier. For example, storage virtualization can map multiple virtual storage devices onto a smaller set of physical storage devices and make them shared by many virtual machines (VMs) [66]. At the same time, it facilitates VM management by providing a variety of functionalities in a virtualized context (e.g., live migration [41, 58], replication [52, 61], consolidation [62, 65], aggregation, metering, server-side caching [35, 37]).

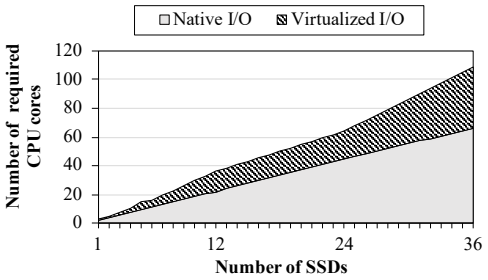


Figure 1: CPU usage of native block I/O in Linux and virtualized block I/O with SPDK vhost-nvme [21]

The importance of storage virtualization is growing for modern datacenters running I/O-intensive big-data workloads on their fast but expensive solid-state drives (SSDs). In particular, it is critical to reduce virtualization overhead and provide near-native storage performance to the VM workloads. A conventional way to overcome the virtualization overhead is to utilize hardware-assisted virtualization mechanisms (e.g., passthrough [30], SR-IOV [19]). However, the existing hardware virtualization mechanisms have become much less appealing to modern datacenters due to their extremely limited VM management support.

To provide highly flexible VM management at minimal virtualization overhead, a new software-based storage virtualization mechanism is now considered as a highly promising solution. A storage performance development kit (SPDK) vhost-target implementation not only enables flexible VM management but also significantly improves performance by exclusively dedicating computing cores (i.e., sidecores) to its user-level virtualization layer [21, 69].

However, such sidecore approaches require a significant

*Corresponding author.

amount of computing resources to execute their polling-based virtual device emulation [53, 59]. Furthermore, the required computing bandwidth quickly increases as a single server is equipped with an increasing number of storage devices, and each device gets faster. As shown in Figure 1, our projection result shows that virtualized I/O with SPDK vhost-nvme [21] necessitates 42% – 65% more CPU cores to saturate multiple Intel Optane SSDs [7] than native block I/O in Linux.

Due to the severe computing resource requirement, the software-based storage virtualization cannot provide high performance or high scalability. First, without enough computing cores dedicated to the storage virtualization layer, the storage system comes to suffer from low performance. Second, without the capability of adding virtualization-dedicated cores as needed, the system comes to suffer from low scalability. Therefore, to achieve high performance, scalability, and flexibility all together, the ideal storage virtualization should decouple itself from host CPU cores, scale with a target storage system, and exploit the most cost-effective computing solution for the programmable VM management.

In this paper, we design and implement FVM, a new hardware-assisted storage virtualization mechanism, to achieve high performance and scalability while maintaining the flexibility to support a variety of VM management features. The key idea of FVM is to implement (1) a storage virtualization layer on an FPGA card (*FVM-engine*) which is decoupled from the host resources, and (2) a hardware-based device-control mechanism to make the card directly manage the physical storage devices. FVM also leverages (3) high-level synthesis (HLS) techniques to provide easy programmability for VM management. Our solution can also be implemented on ASICs for higher performance, but in that case, the ASIC implementations lose future flexibility for new VM management features.

FVM achieves the design goals as follows. First, FVM achieves high performance by utilizing a hardware-assisted virtualization mechanism and leveraging massive parallelism in the modern storage virtualization stack. FVM-engine can cost-effectively exploit the virtualization's parallelism by implementing many wimpy FVM cores and distributing virtual/physical I/O queues and queuing routines to them for fine-grained parallel executions.

Second, FVM achieves high scalability by executing virtual device emulation on FVM-engine, which is decoupled from host CPU cores and device resources. In addition, its direct device-control mechanism further improves the scalability by enabling FVM-engine to directly manage the physical devices. Therefore, without relying on expensive host CPU cores, FVM can achieve highly scalable virtualization performance by implementing FVM-engine on a more powerful FPGA card or adding more FPGA cards on a system board.

Third, FVM achieves highly flexible storage virtualization by implementing existing VM management features on a reconfigurable FPGA card. For user programmability, we leverage an HLS-based design flow and separate the virtualization layer from the I/O logic to interact with the host machine and the physical storage devices.

	Sidecore		PassTh	SR-IOV	FVM
	CPU	On-dev SoC	(1VM)		
	[59, 69]	[53]	[30]	[19]	
Performance†	✔		✔	✔	✔+
Host efficiency		✔	✔	✔	✔
Scalability			✔	✔	✔
Device sharing	✔	✔		✔	✔
Flexibility‡	✔	✔			✔
Programmability	✔	✔			✔

†: I/O throughput, latency, ‡: Providing seamless storage-related services.

Table 1: Comparison of the existing and proposed storage virtualization mechanisms

Table 1 summarizes FVM's key advantages over existing software- and hardware-based storage virtualization mechanisms, in terms of performance, host efficiency, scalability, device sharing, flexibility, and programmability. FVM solves the performance and scalability issues of the recent sidecore approaches, while achieving device sharing and flexibility that the existing hardware-assisted techniques cannot provide. A detailed explanation can be found in Section 3.

For evaluation, we implemented our FVM-engine prototype on a Xilinx FPGA board [23] and Intel Optane SSDs [7]. We implemented Linux device drivers for the software support and augmented an SPDK vhost-target implementation [21] to apply FVM to an existing KVM-based virtualization system [11].

Our experimental results show that the FVM prototype obtains 1.36× higher I/O throughput than the software-based virtualization method when allocating the same amount of host CPU cores. FVM also scales well with the increasing number of VMs and virtual/physical storage devices by achieving 9.5 GB/s aggregate I/O throughput with four SSDs. Also, our HLS-based design flow requires only 10s – 100s of code lines to implement example VM management functionalities.

In summary, we make the following contributions:

- **Novel storage virtualization mechanism**: We propose a novel FPGA-assisted virtual device emulation mechanism for fast, scalable, and flexible storage virtualization.
- **High performance**: FVM achieves high performance by utilizing hardware-assisted virtualization and parallelizing virtual/physical device operations on FVM-engine.
- **High scalability with host efficiency**: FVM can easily increase its computing power to match the target virtualization scalability without depending on host resources.
- **Flexibility & programmability**: Our HLS-based FVM design flow supports easy VM management and feature programmability.

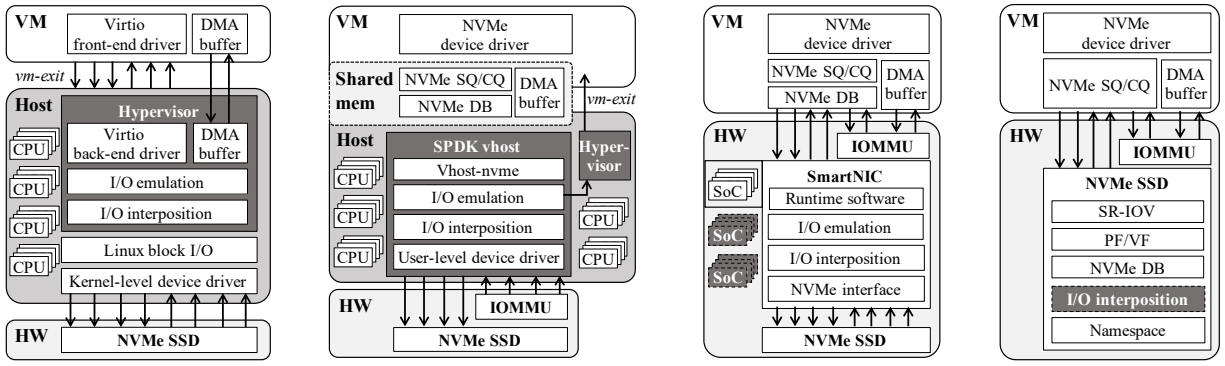


Figure 2: System architectures for conventional storage virtualization mechanisms

2 Background

In this section, we introduce modern non-volatile memory (NVM) technologies and the latest advances in storage virtualization mechanisms.

2.1 NVM and NVMe Protocol

Modern NVM technologies such as 3D XPoint [8] and Z-NAND [18] have significantly improved the storage performance [39, 50, 67, 71]. At the same time, virtualization for such fast storage devices becomes one of the most critical components in cloud environments [53, 55, 59, 69]. For example, Amazon Web Services (AWS) accelerates I/O virtualization through dedicated hardware components [1, 55]. Other major cloud providers, including Microsoft Azure [12] and Google Cloud Platform (GCP) [6], are allowing advanced NVM devices to be used as primary storage for VMs.

NVM Express (NVMe) [14] is a standard storage architecture used to enable fast NVM storage through PCIe and optimized I/O paths. First, it brings multiple deep I/O queues to take full advantage of NVM technologies. The current specification supports up to 65,535 I/O queues, each with 1 – 65,535 outstanding commands. As a result, it can enable highly parallel processing on multiple cores by assigning independent I/O queues and queuing routines to each core or thread. Second, its protocol provides fast I/O submission and completion paths by reducing the number of memory-mapped I/O (MMIO) operations. For example, it does not require MMIO register reads in the common I/O paths, while including a maximum of one MMIO register write for the command submission path.

An NVMe I/O queue consists of a *submission queue* (SQ)/*completion queue* (CQ) pair. For I/O submission, host software places NVMe commands in the SQ and writes the SQ tail pointer to the target *SQ doorbell register* exposed through PCIe base address registers (BARs). The target NVMe storage device then fetches the newly added commands and processes them. Once the NVMe commands are completed, the NVMe device writes completion messages to the associated CQ and then generates an interrupt. Lastly, the host software handles the completion messages and updates the target *CQ doorbell register* to clear the interrupt and release the CQ entries.

2.2 Storage Virtualization

2.2.1 Paravirtualization

In a paravirtualization scheme, a guest operating system (OS) is installed with a VM abstraction to make it efficient to emulate virtual devices. For example, *Virtio* [60] is an abstraction for virtual devices in a hypervisor. This abstraction allows the hypervisor to export a common set of virtual devices and makes them available to guests through an efficient device interface. Figure 2a shows the system architecture for virtio-based device emulation. The guest implements front-end virtio drivers, with particular virtual device emulation behind a set of back-end drivers in the hypervisor [60]. This paravirtualization mechanism can reduce the number of *VM exits* by reducing the number of MMIO operations for the virtual device of the guest, which addresses the huge performance overhead incurred by CPU mode switches and cache pollution [51]. However, the guest OS should be aware that it is being virtualized, which requires modifications to collaborate with the hypervisor efficiently.

Virtio SCSI (*virtio-scsi*) [47] or block (*virtio-blk*) [45] can be used to emulate an NVMe device with this paravirtualization mechanism. They handle VM requests directed at the *virtual* NVMe device as follows: (1) A guest OS makes a request to a virtual device through virtual I/O queues (e.g., *vring* [60]) in virtio front-end drivers. (2) The guest then calls a VM exit and traps into a host machine. (3) The hypervisor emulates the virtual device through virtio back-end drivers, interacting with kernel-level device drivers. (4) Once the I/O request is completed, the virtio back-end drivers read completion messages from the physical devices, confirm their completion status, and inject an interrupt to the guest OS

through the hypervisor.

2.2.2 Host Sidecore Approach

CPU-dedicated (or *sidecore*) approaches can further accelerate storage virtualization by avoiding expensive traps to the hypervisor and reducing cache pollution [33, 48]. The recently proposed SPDK *vhost-scsi* and *vhost-blk* implementations [21] can accelerate virtualization of NVMe storage. As shown in Figure 2b, a hypervisor pre-allocates *shared memory regions* for guests and allows them to exchange storage commands with SPDK vhost-target directly for virtual device emulation. The SPDK vhost-target implementations emulate VM requests as follows: (1) A user-space thread running on a dedicated sidecore continues to poll virtual I/O queues (e.g., NVMe SQ/CQ pairs) via a shared memory region. (2) It reads newly received VM SCSI or block requests and converts them to NVMe commands (i.e., *protocol conversion*). (3) It conducts the I/O operations through an SPDK *user-level* NVMe device driver. (4) Once the requests are completed, another dedicated thread in SPDK vhost-target deals with completion messages and injects an interrupt to the guest through the hypervisor.

The recent sidecore approaches can offer near-native performance of modern NVMe devices to VMs. A dedicated sidecore polls guest I/O operations through shared memory regions, so there is no need to call VM exits to submit NVMe commands. Moreover, SPDK's user-level NVMe device driver enables sidecores to conduct I/O operations without user-kernel mode switches. SPDK vhost-target also reduces the number of data copies by allocating guest DMA buffers in a *pinned* shared memory region. For this, the software-based virtualization layer translates *guest physical addresses* (gPAs) to pinned *host physical addresses* (hPAs). Due to the address translation, the NVMe device can transfer data directly to the guest's memory space without being aware that it receives requests from VMs.

Vhost-nvme. The SPDK *vhost-nvme* implementation [69] further optimizes the sidecore approaches by directly exposing NVMe devices to guest OSes. This transparent view of the NVMe devices can eliminate the performance loss caused by the protocol conversion between SCSI/block and NVMe. It also allows the guest OSes to exploit advanced NVMe features (e.g., shadow doorbell buffer [42]) to get higher performance. A recent study [69] demonstrated that the vhost-nvme implementation gets $1.11\times$ – $1.26\times$ higher random-read throughput than the other SPDK vhost-target implementations.

2.2.3 On-device Sidecore Approach

To save the host resources required for storage virtualization, a recent study [53] offloaded the virtualization layer to system-on-chip (SoC) cores in other peripheral devices (e.g., SmartNIC [2, 13]). Figure 2c shows its system architecture. The *on-device* sidecore approach exposes virtual NVMe interfaces to guest OSes by providing a uniform address space across host CPUs and SoC cores. The SoC allows the runtime software running on the on-device cores to reach virtual NVMe queue pairs mapped in the host memory through DMA. In addition, host software allocates NVMe queue pairs in the SoC's memory space and provides their locations to the physical NVMe device to make it interact with the SoC directly. Since it utilizes on-device sidecores to emulate virtual storage devices, it can save the host CPU resources and offer more compute power to VMs or other VM management features.

Moreover, on-device sidecore mechanisms provide flexible and programmable implementations leveraging ARM-based SoC cores. In particular, it facilitates implementing essential functionalities in storage virtualization, which are not fully offloaded or not easily composable via other hardware-based virtualization mechanisms (e.g., SR-IOV). For example, a recent study [53] implemented storage versioning, prioritization, isolation, replication, and aggregation functionalities in runtime software installed on the SoC.

2.2.4 Direct Device Assignment

To overcome the virtualization overhead, VMs can make use of support for DMA and interrupt remapping (e.g., Intel VT-d [27], AMD-Vi [26]), which allows guest software to access a target storage device directly. For the remapping support, major processor manufacturers introduced *I/O memory management units* (IOMMUs). A DMA remapping engine in an IOMMU allows DMAs from a guest to be accomplished with gPAs. The IOMMU translates them into hPAs according to page tables that are configured by host software. Likewise, an interrupt remapping engine translates interrupt vectors issued by devices based on an interrupt translation table.

The direct device assignment (or *passthrough*) eliminates the virtualization overhead in software layers since the hypervisor is no longer in a guest's I/O paths. However, this approach requires the physical devices to be exclusively assigned to a single VM and does not support device sharing across multiple VMs. Therefore, the passthrough mechanism has limitations in improving the utilization of storage devices and reducing operating costs in modern datacenters.

SR-IOV. To address the challenges of the direct passthrough scheme, the PCIe specification currently supports *SR-IOV* [19], a standardized hardware virtualization protocol. An SR-IOV capable PCIe device supports a *physical function* (PF) and multiple *virtual functions* (VFs). The PF provides resource management for the device and is managed by the host software, and each VF can be assigned to a single VM exclusively for direct access. SR-IOV is now supported by high-performance I/O devices such as network and storage devices as well as accelerators. Recently, Xilinx released an SR-IOV capable PCIe IP block [28], supporting up to 252 VFs.

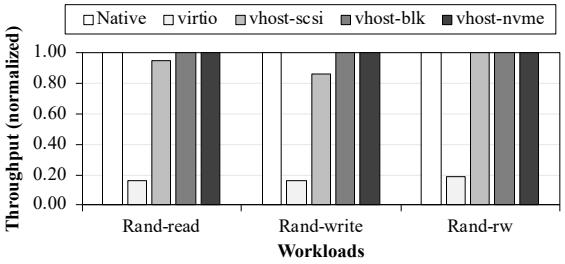


Figure 3: Random I/O throughput with a single SSD

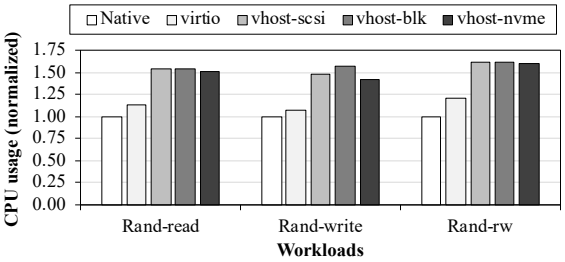


Figure 4: CPU usage of random I/O with a single SSD

Since an SR-IOV capable device implements how to multiplex itself at the hardware level, it does not rely on host software to multiplex the virtual device instances, as shown in Figure 2d. In addition, with SR-IOV and an IOMMU in a host machine, VFs can carry out DMA transactions with gPAs, while avoiding the software-side address translation. Similarly, interrupt remapping for each VF addresses the performance overhead generated by triggering interrupts to notify guests regarding the completion of their I/O requests.

3 Motivation

In this section, we discuss the challenges of the existing virtualization mechanisms for modern datacenters. We identify the critical performance and scalability issues of the existing host and on-device sidecore approaches and the limited VM management support of the hardware-assisted virtualization technologies.

3.1 CPU-inefficient Storage Virtualization

Modern software-based storage virtualization mechanisms dedicate CPU sidecores to emulate virtual NVMe devices. For example, recent NVMe virtualization studies [59,69] allocate multiple CPU cores to poll virtual I/O queues via shared memory regions, instead of making a trap into a host machine. Figure 3 shows the random I/O throughput of the various software-based virtualization implementations on a single CPU sidecore and a single Intel Optane SSD [7], normalized to the native performance. Virtio denotes virtio-based paravirtualization through KVM, and vhost-scsi, -blk, and -nvme mean three different virtual device interfaces through SPDK

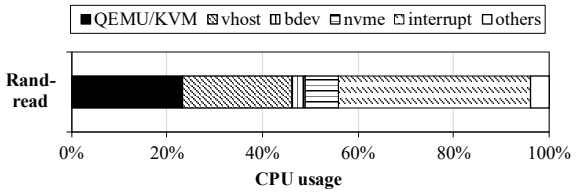


Figure 5: CPU usage breakdown of random-read I/O with SPDK vhost-nvme

vhost-target. We ran FIO [5] random I/O benchmarks with four threads and 32 queue depth and measured the random I/O throughput on VMs. On the other hand, Figure 4 shows the relative CPU usage normalized to that of the native I/O operations on an Intel Xeon server [22] with the same experiment environments. Our experimental results show that virtio fails to offer the full native performance due to frequent VM exits, and all the SPDK vhost implementations can achieve close to the maximum native performance (i.e., 550K IOPS). However, at the same time, to get such near-native performance, they demand $1.42\times$ – $1.61\times$ more CPU resources than native random I/O operations.

There are two primary sources of such high CPU resource usage. First, they utilize multiple active polling cores to reduce the number of VM exits [59, 69]. Because NVMe is a highly parallel storage architecture, the conventional trap-and-emulate approach will generate an unacceptable number of VM exits for a VM to take full advantage of multiple I/O queues [59]. For this reason, the sidecore approaches allocate CPU resources exclusively and poll guest I/O operations through a shared memory region to handle such frequent NVMe requests quickly. Second, the SPDK vhost-target implementations trigger guest interrupts through *eventfd*, which requires system calls and VM exits [69]. Figure 5 shows the CPU usage breakdown of the host machine running SPDK vhost-nvme. Our experimental result demonstrates that around 22% of active CPU cycles are used to poll and emulate virtual devices (vhost) and 39% to trigger guest interrupts (interrupt). The other portions are consumed by the necessary VM management (QEMU [15]/KVM [11]) and the SPDK storage stack (bdev and nvme).

This resource-inefficiency issue poses a significant challenge to scalable storage virtualization and efficient VM management in modern cloud and datacenter environments. To support many NVMe devices and guarantee quality-of-service (QoS) at the same time, the current host sidecore approaches will continue to demand a considerable portion of host CPUs for storage virtualization [44]. Eventually, the number of VMs that can be supported within a single server will decrease, and the total datacenter costs will increase. Otherwise, the VMs will have serious performance problems due to the lack of computing resources. Also, with the limited capability of adding virtualization-dedicated CPU cores per server, the system will come to suffer from the low scalability.

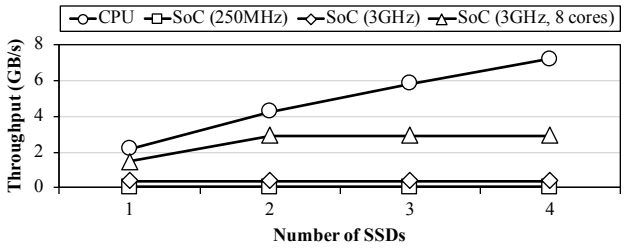


Figure 6: Performance comparison of different virtualization-dedicated core implementations

3.2 Weak Computing Power of SoC Cores

Modern on-device sidecore approaches offload the virtual device emulation to SoC cores embedded in peripheral devices instead of CPU cores to reduce the burden of host CPU [53]. However, their I/O performance can be severely bounded by the SoC cores' weak computing power. To measure the performance bottleneck due to the small cores, we implemented Microblaze softcores (250 MHz) [24] on an FPGA [23] and ran storage virtualization runtime software. We linearly scaled this performance result to evaluate more aggressive SoC designs that use higher clock speed and more cores (3 GHz, 8 cores). Figure 6 shows the random I/O performance of CPU (vhost-nvme) and SoC (runtime software) sidecore approaches with many NVMe devices. Our experimental results show that, even with the number of SoC cores increased, their weak computing capabilities become the significant performance bottleneck.

In particular, SoC sidecore designs significantly suffer from inefficient DMA mechanisms incurred by SW abstraction layers. For example, SmartNICs [2, 13], which utilize SoC cores in NICs, expose RDMA APIs instead of native DMA primitives, and it nearly doubles the DMA read/write latency [56]. Our Microblaze softcore implementation emulated the overhead by adding 5 μs per DMA transaction, and as a result, it achieved only 68% of the maximum performance of a single device. In addition, SoC sidecore designs cannot support a large number of virtual/physical devices and advanced storage management features due to their limited computing capabilities. As shown in Figure 6, an eight-core SoC cannot fully utilize two or more NVMe devices. These scalability issues will become more severe as storage devices get faster.

3.3 Absence of Interposition Layer

To save the host and on-device sidecores, hardware-assisted virtualization techniques can bypass the host software entirely. We experimentally confirmed that the two popular HW-assisted virtualization technologies in modern NVMe SSDs (i.e., passthrough and SR-IOV) provide the near-native performance in VMs. For this purpose, we installed a Samsung PM1733 SSD [17] which offers both passthrough and SR-IOV capabilities, and measured its FIO random I/O performance in VMs. We created a single VF through SR-IOV and assigned a 128-GB namespace, eight virtual queues, and eight virtual interrupt resources. When the device is connected through PCIe Gen3, we obtained around 800k IOPS for random reads and 250k IOPS for random writes in both passthrough and SR-IOV environments.

However, they suffer from the limited VM management and storage features in cloud environments. For example, SR-IOV does not support critical features to enable easy storage management such as live migration [41, 58] and seamless switching between different I/O channels. Also, it does not allow hypervisors to add critical features that are not natively provided by physical devices: replication [52, 61], snapshot [40, 70], record-replay, deduplication [68, 72], compression, encryption [63], metering, accounting, billing, and throttling [36, 46, 54] of guest I/O activities.

In addition, such hardware techniques enabling only the specific in-storage features significantly limit their portability and fungibility in modern datacenters. Furthermore, their fixed and vendor-specific storage functionalities do not provide enough flexibility to support advanced VM management. It is still challenging to provide flexibility and high performance at the same time with the current hardware-assisted virtualization schemes.

4 FVM Design and Implementation

This section introduces the design goals for fast, scalable, and flexible storage virtualization, and proposes our FVM solution to satisfy the goals. We describe our solution by presenting (1) a front-end implementation that emulates virtual devices and (2) a back-end implementation that directly manages physical devices.

4.1 Design Goals

We set the following design goals to resolve the challenges in modern storage virtualization: (1) A next-generation virtualization mechanism should ensure the near-native performance of NVMe storage devices. (2) It should minimize the amount of host resources used for virtualization so that a host machine can provide more computing power to VMs. (3) At the same time, it should nicely scale with the number of storage devices. (4) A physical storage device should be shared by multiple VMs. (5) It should not rely on hard-wired units to enable flexible and essential management functionalities, as summarized in Table 2. For example, software-based virtualization can implement flexible VM management features, while SR-IOV makes it hard for system administrators to guarantee accurate feature behaviors as in-SSD resource allocation and scheduling are done in a vendor-specific way.

Category	Features	SW	SR-IOV	FVM
Storage configuration	Consolidation	✔	✔	✔
	Aggregation	✔		✔
	Caching	✔		✔
Resource management	Isolation	✔	△	✔
	Throttling	✔	△	✔
Fault tolerance	Replication	✔	△	✔
	Snapshot	✔	△	✔
Data manipulation	Compression	✔	✔	✔
	Deduplication	✔	△	✔
	Encryption	✔	✔	✔
Administration	Migration	✔		✔
	Metering	✔	△	✔
	Billing	✔	△	✔

△: Limited to single-device use cases.

Table 2: Example VM management features in storage virtualization layers

4.2 FPGA-assisted Storage Virtualization

To meet the design goals, we propose FVM, a new hardware-assisted storage virtualization mechanism. The key idea of FVM is to implement *FVM-engine*, an FPGA-based virtualization acceleration card. We implement a storage virtualization layer and a device-control mechanism on FVM-engine.

In contrast to on-device SoC cores, an FPGA can be configured only with essential elements for storage virtualization and can take advantage of highly parallel NVMe protocols. Our FPGA-based solution can implement many cost-effective cores, and distribute virtual and physical NVMe queues and management routines to the cores. In this way, our solution achieves fine-grained parallel executions and scalable performance. In addition, our solution uses an FPGA's on-chip memory for SQ/CQ pairs and doorbell registers, which can be fast and directly accessed by VMs and NVMe devices through PCIe.

Another advantage of our FPGA-based solution is its programmability to implement new VM management features. Our FPGA-based solution has the potential of the hardware-based virtualization to solve the performance and efficiency challenges, while allowing to implement various VM management functionalities with its reconfigurability. In this work, we propose an FPGA-based virtualization layer, but it is also possible to implement the mechanism on ASICs. In such a case, an ASIC implementation can achieve higher performance by leveraging its optimized circuits for virtualization functions, but its flexibility for new storage management features will be limited.

Figure 7 shows the FVM architecture and its components. First, FVM bypasses host software stacks entirely and minimizes the use of host resources. Through the SR-IOV implementation on FVM-engine, VMs can enter a virtualization layer without any arbitration support from the host software. Moreover, its hardware-level NVMe interface makes the card directly manage the physical NVMe devices through PCIe.

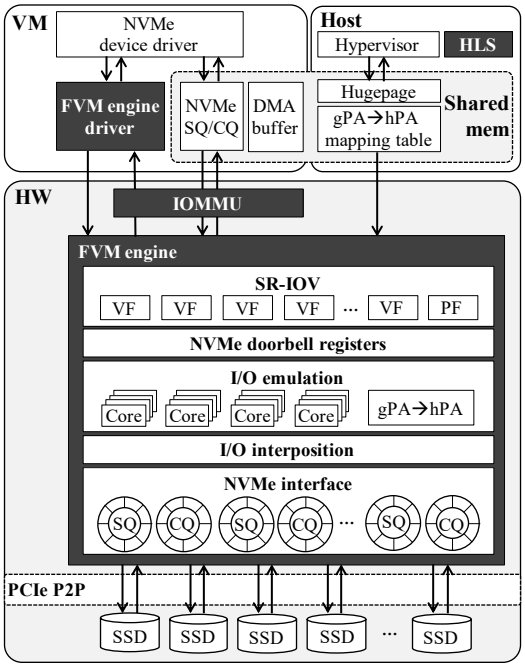


Figure 7: FVM architecture

Second, FVM is able to scale with many NVM devices by employing a parallel architecture for the device emulation. Instead of relying on on-device SoC cores, FVM-engine incorporates many specialized hardware units to poll and emulate guest I/O operations. Third, its HLS-based design flow enables flexible and programmable implementations for FVM-engine and other storage management services.

4.3 Front-end: VM-to-FVM-Engine

Direct FVM-engine assignment. FVM assigns virtual instances of FVM-engine to each VM through its SR-IOV interface. The current FVM-engine implementation integrates a PCIe IP block [28] to enable its own SR-IOV interface and supports up to four physical functions (PFs) and 252 virtual functions (VFs). The PFs are managed by host software for resource management, and each VF is assigned to a single VM exclusively for direct access to FVM-engine. Since all VFs have an identical PCIe configuration (e.g., PCIe BAR), VMs can install the same guest FVM-engine driver. FVM-engine also successfully isolates MMIO from different VMs by applying *non-overlapping* address translation to its internal address space (e.g., PCIe-to-AXI address translation [28]). At the same time, with the IOMMU support, FVM-engine can perform DMA transactions to guest memory space and inject an interrupt without a host software arbitration. To enable such *exitless* DMA transactions and interrupts, we install Linux *virtual function I/O* (VFIO) drivers in the host machine.

There are three major benefits of providing SR-IOV in FVM-engine. First, this design enables CPU-efficient virtual device emulation. All VMs can directly enter this hardware

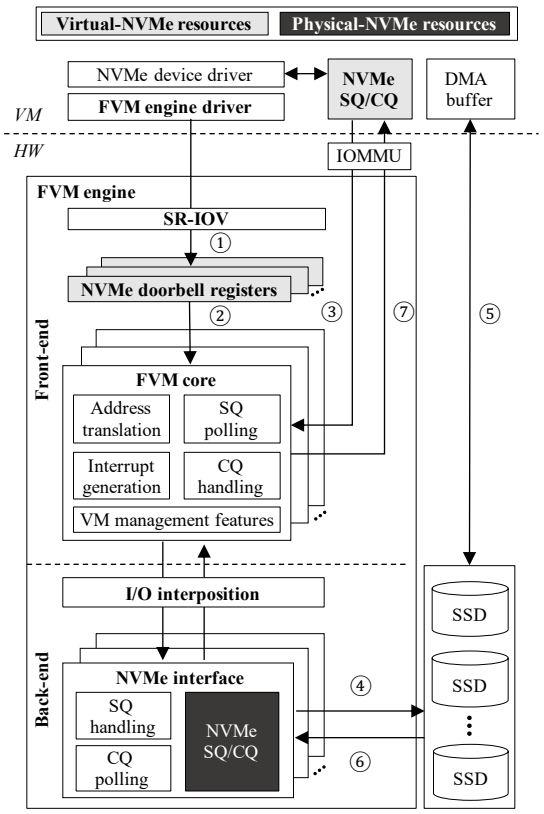


Figure 8: Hardware-level device emulation mechanism

Algorithm 1: Polling function in the front-end for I/O submission

```
1  fvm_nvme_submit (devices_virtual);
2  while true do
       /* Iterate over the assigned virtual
          devices and their SQs                    */
3      foreach vdev ∈ devices_virtual do
4          foreach sq ∈ available_sqs(vdev) do
               /* Poll the doorbell registers mapped
                  in FVM-engine                    */
5              tail = get_tail(sq)
               /* Find newly added NVMe commands
                  from the guest OS                */
6              head = get_head(sq)
7              while tail ≠ head do
8                  cmd = get_cmd(sq, head)
9                  cmd = manipulate_cmd(cmd)
10                 submit_cmd(cmd)
11                 head = (head + 1)%SQ_SIZE
12             end
13             set_head(sq, head)
14         end
15     end
16 end
```

interposition layer and handle interrupts without host software intervention. Second, it allows multiple VMs to share FVM-engine through 252 VFs. Using this interposition layer, FVM-engine can map virtual devices onto a much smaller set of physical NVMe devices. Third, it does not rely on fixed or vendor-specific storage capabilities. By simply deploying FVM-engine, any host machine can benefit from this virtualization mechanism.

Doorbell register remapping. FVM-engine reserves a memory space for NVMe doorbell registers and exposes it through PCIe BARs. When guest NVMe device drivers call `nvme_write_sq_db()` to submit I/O requests, the guest FVM-engine driver intercepts them and obtains their *(virtual) device id*, *SQ id*, and *SQ tail* information. The guest FVM-engine driver then calculates the address of the target doorbell register and writes the received SQ tail pointer to FVM-engine (Figure 8–①). In this way, the guest OS can indicate new NVMe commands to be executed. Similarly, to notice that the command completions are normally handled, FVM-engine driver intercepts `nvme_process_cq()` function and acquires *(virtual) device id*, *CQ id*, and *CQ head* information. It then writes the received CQ head pointer to the target address in the FVM-engine doorbell regions.

Virtual I/O queue emulation. To process a guest I/O request at the hardware layer, FVM-engine polls doorbell registers using multiple FVM cores (Figure 8–②). Algorithm 1 demonstrates its polling routine to emulate virtual NVMe devices. First, the FVM core gets the newly updated `SQ tail` (line 5) and compares it with the `SQ head` that stores the previous `tail` value (line 6). The difference between these two values indicates the number of commands that are submitted by the guest OS. Since FVM-engine reserves its doorbell memory regions using on-chip memory (e.g., BRAM [29]), it can quickly poll those regions. This design can easily scale up the number of VMs as modern FPGAs currently support tens of MBs on-chip memory [23].

To enable FVM-engine to access a submission queue in the guest memory space, we utilize an internal DMA engine [28] and an IOMMU. When VMs install guest NVMe drivers, they deliver SQ/CQ gPAs to FVM-engine. FVM cores then use these addresses to directly read the submitted commands through the DMA engine (Figure 8–③). Since FVM guarantees exitless DMA transactions with an IOMMU and VFIO drivers, each virtual instance of FVM-engine can safely access target SQ/CQ pairs allocated in the guest memory space without software intervention.

Similarly, to deliver NVMe completion entries to a VM, FVM-engine directly writes the completion messages to the guest CQ memory region (Figure 8–⑦). In addition, it triggers an interrupt to the guest directly through the interrupt remapping engine. The FVM-engine driver then forwards the interrupt with an associated IRQ vector to the NVMe driver and allows it to handle the received completions.

PRP and LBA translation. FVM-engine processes the received NVMe commands from VMs before submitting them

to physical NVMe devices. Specifically, FVM-engine manipulates *physical region page* (PRP) entries (pointing guest DMA buffers) from gPAs to hPAs. To enable such gPA-to-hPA translation at the hardware level, FVM leverages *hugepages* to allocate pinned memory [4, 20]. Since the current operating system does not change their physical locations, FVM-engine can statically translate PRP entries by incorporating the gPA-to-hPA mapping table. The translation does not incur any performance overhead in this design as FVM-engine manages the mapping table using its on-chip memory. Also, due to the hugepages (2MB), the required table size is small enough to keep them in the on-chip memory (i.e., 4KB table to cover 1GB guest memory space).

In addition, FVM-engine needs to manipulate a *start logical block address* (SLBA) to allocate separate block regions of physical devices to VMs. Since the current implementation of FVM assumes a *static* partition, the SLBA in guest NVMe command can be simply modified by applying a different offset value, which is managed by host software.

Virtual admin queue emulation. FVM manages a virtual NVMe *admin* SQ/CQ pair through QEMU and SPDK vhost-target implementations. Since QEMU and KVM can track VM exits caused by MMIO on administration doorbell registers, they are still able to interact with SPDK vhost-target via a UNIX domain socket. QEMU delivers critical administration commands (e.g., I/O queue creation, deletion, shutdown) to the SPDK vhost-target implementation following the conventional vhost-target protocol.

4.4 Back-end: FVM-Engine-to-SSD

Physical SQ/CQ remapping. To allow FVM-engine to interact with physical NVMe devices directly, host software remaps their NVMe I/O queues onto FVM-engine's PCIe BAR regions. At the installation time, the host FVM-engine driver provides the memory-mapped region's address to the physical NVMe devices. The NVM devices are unaware of FVM-engine, but a PCIe switch delivers DMA transactions to FVM-engine seamlessly. Also, our experimental result demonstrates that FVM-engine can fully utilize a single Intel Optane SSD with eight SQ/CQ pairs (4KB each queue). Thus, FVM-engine can nicely scale with a large number of physical devices and VMs without any on-chip memory space issue for these remapped queues.

Direct NVMe device-control mechanism. FVM-engine incorporates standard NVMe interfaces to implement a direct device-control mechanism. (1) FVM-engine moves the NVMe commands to the submission queue in the FVM-engine on-chip memory. (2) FVM-engine then rings doorbell registers located in the NVMe device to notify the number of newly submitted commands. (3) The NVMe controller fetches the NVMe commands through PCIe P2P communications (Figure 8–④). (4) After the NVMe device processes the commands (Figure 8–⑤), it writes the command completions to the FVM-engine address space (Figure 8–⑥). (5) FVM-engine processes them and (6) rings doorbell registers located in the NVMe device.

Polling CQs. To immediately handle completions from physical devices, an NVMe interface polls its CQ memory space. Algorithm 2 shows its polling function. First, the NVMe interface handles a CQ entry pointed by its `head` pointer (line 7) and compares its `phase` bit with the current round (line 9). This enables the NVMe interface to determine whether a new entry was posted as a part of the previous or current round of completion notifications. After that, it processes the completion entries (line 10) and forwards them to the front-end (line 11). The FVM core then writes completion messages to the guest CQ memory space. Since FVM-engine manages all SQ/CQ pairs using the on-chip memory, its polling routine does not incur any performance overhead.

Algorithm 2: Polling function in the back-end for I/O completion

```
1  fvm_nvme_complete (devices_physical);
2  while true do
       /* Iterate over the assigned physical
          devices and their CQs                    */
3      foreach pdev ∈ devices_physical do
4          foreach cq ∈ available_cqs(pdev) do
5              head = get_head(cq)
6              cq_phase = get_cq_phase(cq)
               /* Poll the completion entries mapped
                  in FVM-engine                    */
7              cpl = get_cpl(cq, head)
8              cpl_phase = get_cpl_phase(cpl)
               /* Find newly added NVMe completions
                  from the physical device         */
9              while cpl_phase == cq_phase do
10                 cpl = manipulate_cpl(cpl)
11                 forward_cpl(cpl)
12                 head = (head + 1)%CQ_SIZE
13                 if head == 0 then
14                     cq_phase = invert_phase(cq_phase)
15                 end
16                 cpl = get_cpl(cq, head)
17                 cpl_phase = get_cpl_phase(cpl)
18             end
19             set_head(cq, head)
20             set_cq_phase(cq, cq_phase)
21         end
22     end
23 end
```

4.5 FVM Core Design

FVM maximizes the opportunities of its hardware-level virtualization mechanism by instantiating multiple FVM cores to poll and emulate guest I/O. This design choice can offer more

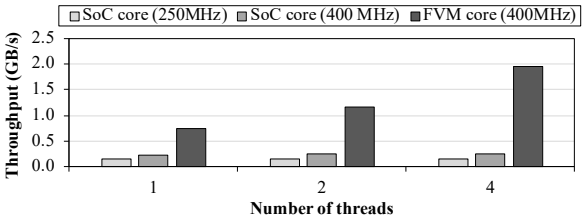


Figure 9: Performance comparison of different virtualization processing core implementations

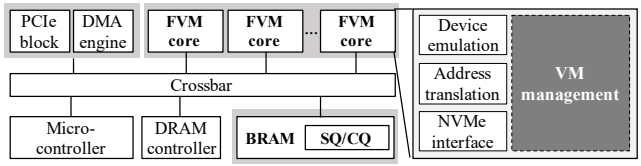
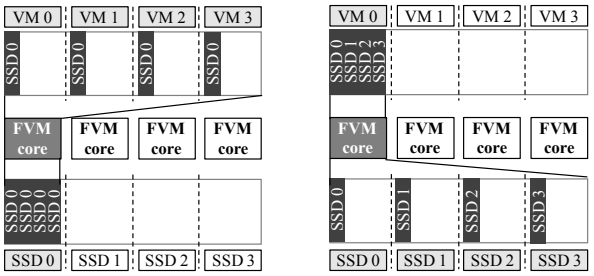


Figure 10: Multiple FVM cores with crossbar interconnect

scalable performance than an on-device SoC core as it replaces the general-purpose sidecores with the customized hardware units. By doing so, FVM can address the performance bottleneck in the processing cores. Figure 9 shows the performance difference between *single-core* SoC and *single-FVM core* implementations. To measure the performance bottleneck on the SoC cores, we implemented runtime software on Microblaze softcores [24] and ran FIO random read benchmarks with the increasing number of threads. We projected this performance result for the higher clock speed (400 MHz) to fairly compare it with our FVM core implementation running at the same clock frequency. Our experimental result demonstrates that the current FVM core implementation achieves 8× higher throughput than the softcore implementations.

Figure 10 shows an example system architecture using multiple FVM cores for storage virtualization. First, to reduce the burden on users in building and integrating system functions required for interacting between VMs and NVMe devices, we separate the virtualization logic (storage service) from the common I/O (BRAM, crossbar) and the board-specific logic (DRAM, PCIe). Second, by interconnecting them with crossbars, FVM-engine can be extended to support a multi-core



(a) Multi-VM support (b) Multi-SSD support

Figure 11: Multi-VM and multi-SSD support with FVM cores

Designs	LUTs	Registers	BRAMs	Clock speed
1 core	4682 (0.36%)	7528 (0.29%)	5.5 (0.29%)	400 MHz
1 VM - 6 cores (5 SSDs)	106809 (8.19%)	130064 (4.99%)	362.5 (17.98%)	400 MHz
4 VMs - 6 cores (4 SSDs)	103539 (7.94%)	131394 (5.04%)	359.5 (17.83%)	400 MHz
4 VMs - 2 cores (1 SSD)	80259 (6.16%)	93997 (3.61%)	321.5 (15.95%)	400 MHz

Table 3: FPGA resource utilization for different FVM configurations

design. Our crossbar interconnection can have 16 input/output ports and can be connected with other switches for higher scalability. As each crossbar switch takes tens of nanoseconds, the overall switching latency is negligible compared to modern SSD's microsecond-scale access latency.

In addition, FVM-engine can be configured to support various FVM core mapping strategies. For example, Figure 11 shows two different mapping strategies. Figure 11a demonstrates that a single FVM core is shared by multiple VMs, while it is dedicated to a single NVMe device. This design can easily cover an increasing number of VMs. On the other hand, Figure 11b shows that an FVM core is dedicated to a single VM, while it covers multiple physical NVMe devices. With this mapping strategy, we can allocate more virtualization resources to more performance-critical VMs.

As Table 3 shows, FVM can cost-effectively scale with multiple FVM cores without sacrificing its clock speed. We implemented three different FVM configurations depending on the number of VMs and SSDs that can be supported. The 1-VM and 6-FVM-core implementation supports five NVMe SSDs and utilizes 8.19% LUTs, 4.99% registers, and 17.98% BRAMs in the FPGA chip. Also, its light resource usage (< 0.5% for a single FVM core) provides opportunities to utilize the remaining resources to implement more FVM cores, and/or deploy much cheaper FPGA boards to minimize the FPGA costs. In addition, FPGAs and FVM cores can be more easily added/upgraded on servers, which provides higher scalability using expandable slots than CPU cores requiring extra sockets.

4.6 HLS

FVM enables flexible and easily programmable implementations through its high-level synthesis (HLS)-based design flow. Modern HLS supports high-level languages and has become a standard hardware design flow for FPGAs. Our HLS-based FVM implementation allows users to extend their designs easily.

In this work, we implemented five different storage functions on FVM-engine First, we implemented device sharing, which allows multiple VMs to share a single NVMe device. Second, we designed a token-based throttling mechanism to effectively manage guest I/O operations. Third, we implemented replication to achieve fault tolerance. Fourth, we

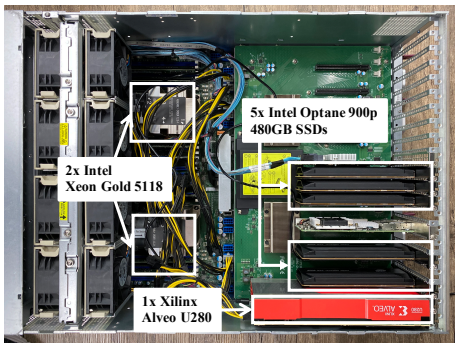


Figure 12: FVM hardware prototype

implemented server-side caching to accelerate storage accesses from VMs. Fifth, we designed direct copying in which a guest OS moves data between two different storage devices using FVM-engine. For this purpose, the VM utilizes FVM-engine's internal memory space as an intermediate buffer, while bypassing the entire software stacks.

5 Evaluation

In this section, we evaluate our FVM implementation and compare its random I/O and RocksDB performance with other storage virtualization schemes. We also present five example VM management features implemented through our HLS-based design flow.

5.1 Experimental Setup

To evaluate FVM, we ran FIO [5] random I/O benchmarks and RocksDB [16] workloads on VMs. We evaluated our FVM implementation against its native execution and existing virtualization mechanisms including SPDK vhost-nvme v20.01 [21] (configured with the option `-with-internal-vhost-lib`) and *passthrough*. Since the passthrough technique avoids most of the virtualization software stack and directly assigns the device to the VM, it can provide the near-native execution performance. As FVM's use cases, we also implemented five different storage services (device sharing, throttling, replication, caching, and direct copy) based on our HLS-based design flow, and validated them with respect to the software reference implementations.

Figure 12 shows our hardware FVM prototype. We built this prototype on a host machine (Super Micro SuperServer 4029GP-TRT2) with two 12-core Intel Xeon Gold 5118 CPUs running at 2.3GHz, 256GB DDR4 DRAM, and five 480GB Intel Optane 900P NVMe SSDs. The Optane SSD (based on the 3D XPoint NVM technology) can support up to 550k IOPS in random-read and 500k IOPS in random-write with 10 μs latency [7]. We implemented FVM-engine on a Xilinx Alveo U280 Data Center Accelerator Card using Vivado and Vivado HLS v2019.2 EDA tools. We configured the PCIe

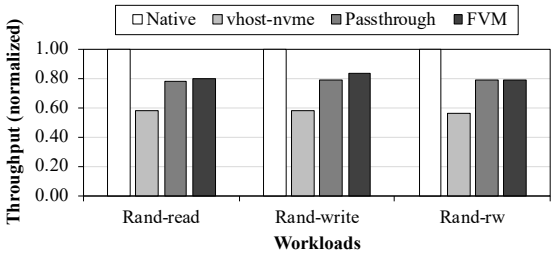


Figure 13: Random I/O throughput with two NVMe SSDs

IP block to meet the PCIe Gen3 x4 specification and connected it with other NVMe SSDs through PCIe Gen3 x16 lanes. For accurate performance measurements, we disabled hyperthreading and dynamic voltage and frequency scaling (DVFS).

On the software side, we installed 64-bit Ubuntu 18.04 with the Linux kernel version 5.3.0 and QEMU emulator version 3.0.0 on the host machine. We installed the same OS and Linux kernel versions on VMs, and implemented an FVM-engine Linux device driver. We modified an SPDK vhost-target implementation and applied FVM to an existing QEMU/KVM virtualization system.

5.2 Performance

5.2.1 Random I/O Benchmark

To evaluate the random I/O performance, we ran FIO with two SSDs and measured (1) the maximum achievable throughput, (2) latency, and (3) CPU utilization. For passthrough and FVM, we allocated four CPU cores and 1GB system memory per VM. To show the performance impact due to the lack of host resources in vhost-nvme, we allocated one CPU core for the vhost-nvme virtualization layer and three cores for the VM.

Figure 13 shows the relative throughput of 4KB random read, write and read/write (50% of read and write each) for native, SPDK vhost-nvme, passthrough, and FVM. For all three random I/O benchmarks, passthrough and FVM can achieve about 79% (2.65GB/s on average) of native performance (3.36GB/s on average). However, SPDK vhost-nvme achieves about 58% (1.95GB/s on average) due to the CPU resource competition between VMs and the vhost-nvme virtualization layer.

In this experiment, we observed that other virtualization overheads still prevent even the passthrough and FVM from achieving the full native performance. First, passthrough and FVM include VM exits caused by MSR_WRITE and HLT instructions to manage timer interrupts and to yield CPU resources to a host machine. Second, they involve IOMMU's address translation to transfer data to and from NVMe storage directly (passthrough) or to manage guest OSes' SQ/CQ pairs from FVM-engine (FVM). There have been efforts to mini-

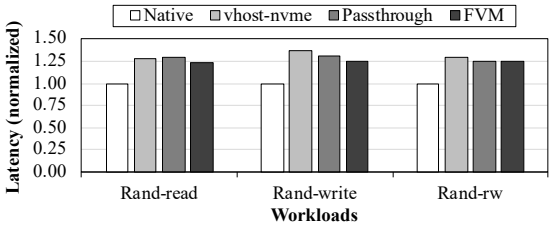


Figure 14: Random I/O latency with two NVMe SSDs

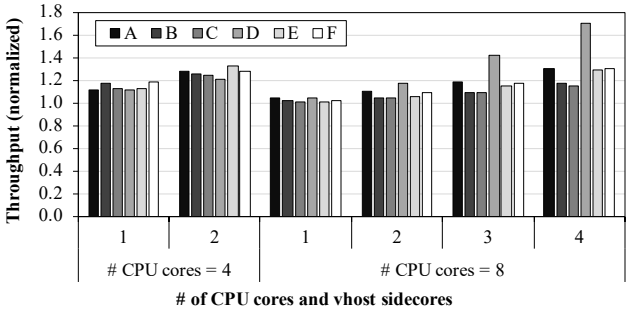


Figure 15: RocksDB throughput with FVM (normalized to SPDK vhost-nvme)

mize the overheads [34, 57]. In this work, we do not address the overhead as they are orthogonal to our work.

In addition, we can see that FVM performs better than passthrough in some cases. Our current FVM-engine implementation aggregates completions from those two SSDs and delivers the smaller number of interrupts to the VM, which provides more CPU resources to random I/O operations.

Figure 14 shows the average latency normalized to that of the native execution for the FIO experiments. For all three workloads, FVM outperforms vhost-nvme and passthrough thanks to the fast FVM core design and the direct device-control mechanism through PCIe P2P communications.

5.2.2 RocksDB

To evaluate a server workload on FVM, we ran RocksDB [16] on the EXT4 file system and YCSB [38] to generate workloads. We configured YCSB to generate the workloads as follows: (A) 50% of read and write each, (B) 95% of read and 5% of write, (C) read-only, (D) read-latest (most reads access the last write), (E) short-ranges (most reads access recent writes), and (F) read-modify-writes. We scaled up RocksDB's `recordcount` and `operationcount` parameters to highlight its I/O activities.

For SPDK vhost-nvme, we considered various CPU allocation scenarios and increased the portion of dedicated sidecores up to 50% to emulate future high-performance NVMe devices. For this purpose, we assigned 1 – 4 CPU cores (out of 4 or 8) for SPDK vhost-target and the remaining CPU cores for the VM. For FVM, we assigned four or eight CPU cores for the VM.

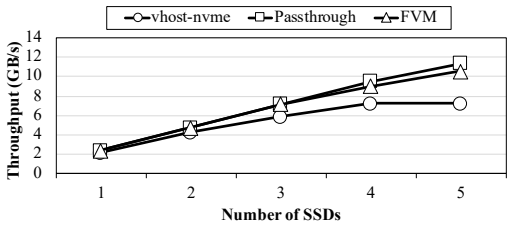


Figure 16: I/O throughput with multiple SSDs

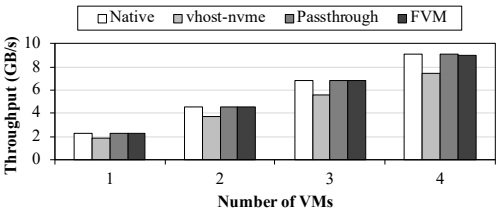


Figure 17: Multi-VM throughput with multiple SSDs

Figure 15 shows the operation throughput of FVM, normalized to that of SPDK vhost-nvme. Since FVM saves host CPU resources to provide more computing power to VMs, it obtains 1.20× (average) higher and 1.33× (maximum) higher throughput than vhost-nvme with four CPU cores. On the other hand, with eight total CPU cores, FVM achieves 1.15× (average) higher and 1.71× (maximum) higher throughput than vhost-nvme. With these trends, FVM will become more promising as the storage devices get faster in future.

5.3 Scalability

For the scalability test, we installed one VM with 18 CPU cores to fully utilize five NVMe SSDs and measured the aggregate throughput. For vhost-nvme, we assigned four cores to SPDK vhost-target and 14 cores to the VM. Figure 16 shows the total throughput that a single VM can achieve with the given number of SSDs. As the number of SSDs increases, both passthrough and FVM scale nicely, while vhost-nvme does not scale well due to its excessive CPU usage to emulate the guest I/O operations in the hypervisor.

When the VM utilizes more than four SSDs, FVM's total throughput is 7% lower compared to that of passthrough. Because the current PCIe IP core [25] supports only eight interrupt vectors per VF, the FVM-engine device driver installed in the guest OS should make the interrupt vectors shared by many SQ/CQ pairs and look up multiple CQs to identify completion messages.

Next, we assigned four CPU cores and one SSD to each VM and ran four VMs concurrently. For vhost-nvme, we assigned one CPU core to SPDK vhost-target and three cores to the VM. Figure 17 shows the total achievable throughput of each virtualization implementation as the number of VMs increases. The results show that FVM scales well as the number of VM increases. With four VMs, FVM achieves up to 9.5

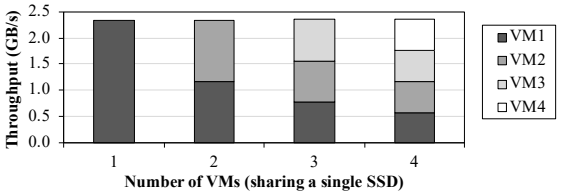


Figure 18: Device sharing with balanced allocation

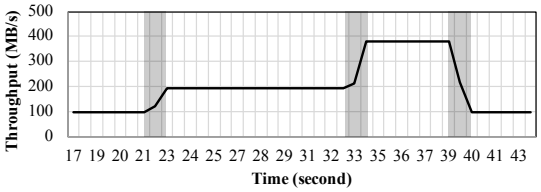


Figure 19: SSD throughput trace with token-based throttling

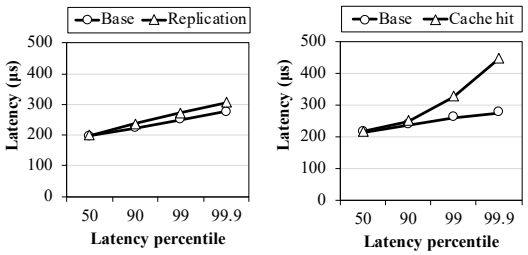
GB/s by processing on FVM-engine. However, vhost-nvme fails to achieve the full native throughput, due to its software overhead.

5.4 Programming Example Functions

To evaluate FVM's flexibility, we implemented five example storage functions in FVM's hardware-level virtualization layer: (1) device sharing, (2) throttling, (3) replication, (4) caching, and (5) direct copy. To implement these functions, we added and modified only 10s-100s of C++ code lines (i.e., device sharing: 40 LOC, throttling: 70 LOC, replication: 15 LOC, caching: 220 LOC, direct copy: 570 LOC).

Device sharing. To implement the device sharing functionality, we mapped multiple virtual I/O queue pairs from different VMs to a single physical NVMe queue pair (similar to SPDK vhost-nvme implementations). To correctly arbitrate completion messages from the physical device, we made an additional data structure to keep track of the virtual device identifications of the submitted NVMe commands which requires 64 bytes for each physical I/O queue. We also deployed a round-robin method between virtual I/O queues as the vhost-nvme implementation does. Figure 18 shows the FIO throughput results of FVM when running multiple VMs on a single NVMe SSD. FVM achieves the perfectly balanced throughput allocation among VMs without any performance loss.

Throttling. We implemented a token-based throttling algorithm on FVM, which can limit the bandwidth with periodically refilled tokens and a bucket which can save a certain amount of tokens. The FVM-engine driver configures the period of `refill_token` signal, the amount of tokens to be refilled in a period, and the size of the bucket. An FVM core periodically polls the `refill_token` signal and filters every command by checking the size of the request and the amount of remaining tokens. If the command is issued, a proper amount of tokens are removed from the bucket. Figure 19 shows the aggregate bandwidth within a time interval of the FIO benchmark while throttling the I/O operations on FVM-engine. We configured the FIO benchmark to achieve the maximum bandwidth of an SSD (2GB/s), and we throttled the I/O from 100MB/s to 400MB/s. The figure shows that the bandwidth is stable and limited as configured.



(a) Replication - write (b) Cache - read

Figure 20: Tail latency of replication and caching

Replication. By seamlessly replicating I/O operations from VMs, a server with FVM-engine can achieve fault tolerance. To enable this feature, we assigned multiple physical NVMe SSDs to a single virtual storage device. When an FVM core receives a write command from a VM, it replicates the commands and broadcasts them to the physical devices. The FVM core then waits for completion messages from all physical storage devices assigned to the virtual device before sending corresponding completions to the target VM. Figure 20a shows its tail latency results of 4KB random writes through FVM. As an FVM core should replicate NVMe commands and wait completions from all the NVMe SSDs, the replication feature adds the extra latency compared to the baseline FVM implementation.

Caching. To enable caching, we implemented a hash table that has 256k entries in the FPGA's 4MB on-chip memory space. Each table entry contains a start logical block address (SLBA) and a corresponding cached block address (CBA). If an FVM core finds a valid entry in the hash table, it replaces the received SLBAs with CBAs and submits them to the NVMe devices. We measured the tail latency of 4KB random reads with FVM and its caching mechanism, while emulating a perfect cache hit ratio. Figure 20b shows that the caching mechanism increases the tail latency due to the increased number of contentions on FVM-engine's interconnection resources for the hash table accesses.

Direct copy. FVM can enable a direct device-to-device (D2D) data copy feature, while bypassing the host CPU and memory. Using the feature, a server with FVM-engine can perform intra-VM or inter-VM data transfers efficiently. We implemented a direct-copy feature on FVM leveraging its hardware-based device-control mechanism. When FVM-engine receives a request, it splits the bulk data transfer into multiple smaller-sized requests. It then generates NVMe commands for each

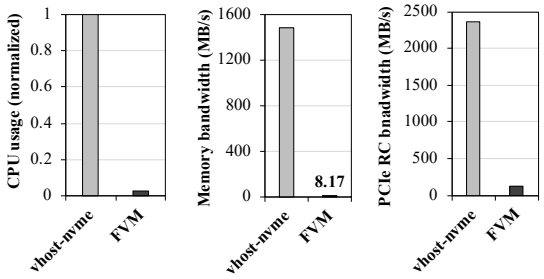


Figure 21: Resource usage of direct copying

split request and submits them using FVM-engine's NVMe interface. To bypass the host memory, the NVMe commands utilize the FPGA's on-chip memory as an intermediate data buffer. Figure 21 shows the CPU usage, host memory bandwidth, and PCIe root complex (RC) bandwidth usage while performing a 32GB inter-SSD bulk data transfer. This figure shows that the data transfer controlled by FVM provides high bandwidth without consuming the host resources.

6 Discussion

Cost analysis. FVM can be used for the cost saving, as it minimizes the number of the required CPUs and servers for the target storage virtualization. For example, for a server with 24 NVMe devices, FVM can reduce the CPU core usage by up to 30% (i.e., saving 20 cores in a 64-core machine). Based on the current prices on major online stores (e.g., Amazon) and vendor websites, FVM can save $2000–$6400 ($100–$320 per core [9,10]) for 64-core machines. If we consider the trend of increasing the number and performance of storage devices per server, the cost saving will be even more significant. In addition, as our wimpy FVM core uses very small FPGA resources (< 0.5%), we can implement FVM on the cost-effective FPGA boards (e.g., $3000 for Alveo U50 [3], $1500–$3500 for evaluation boards).

FVM-engine scalability. The scalability bottleneck can occur if an FPGA is short of resources and/or communication takes too long. On our FPGA, we can implement around 140 wimpy FVM cores (0.5% per FVM core) including multi-level crossbar networks (2.5% per crossbar module) and hundreds of SQ/CQ pairs (5KB per pair). Our crossbar switch can have 16 input/output ports and can be connected with other switches. As each crossbar switch takes 24 ns (6 cycles, 250MHz), the overall switching latency is negligible compared to modern SSD's tens of microsecond access latency.

Supporting other storage protocols. One of the biggest benefits of using programmable FPGAs is to provide various storage protocols as needed. The FPGA-based virtualization layer can easily implement a new interface to activate advanced features in modern storage devices. For example, it can easily support standardized key-value store (KV) acceleration extensions [43] by reprogramming the FPGA according to their interface specifications.

7 Related Work

NVMe virtualization. NVMe virtualization requires a special mechanism to make full use of its parallel and high-performance storage protocol. SPDK [20] is a user-space library for high-performance and scalable storage applications. It integrates all the necessary drivers into the user space to avoid system calls and enable zero-copy access from the applications. In addition, it adopts polling to monitor I/O completions instead of relying on interrupts. Specifically, SPDK vhost-nvme [69] extends the SPDK library to provide virtual NVMe controllers to QEMU-based VMs. Similarly, MDev-NVMe [59] provides a mediated passthrough mechanism in kernel space with an active polling mode.

Direct device-control mechanism. A direct device-control mechanism at the hardware level provides fast and resource-efficient I/O paths. For example, device-centric server (DCS) and its direct device-control method [32, 49] implement a device orchestration scheme on an FPGA to enable fast device-to-device direct data communications. In this way, DCS can enable hardware-offloaded direct data transfers between NVMe SSDs and network adapters through PCIe P2P. As another example, GPUDirect Async [31] enables a direct data transfer between GPUs and NICs to free CPUs from the control path, while moving data between GPUs and NICs. Lynx [64] offloads the server data and control planes to a SmartNIC, and enables direct networking from accelerators via a lightweight hardware-friendly I/O mechanism. It enables to develop hardware-accelerated network servers which do not require much CPU involvement.

8 Conclusion

In this work, we present FVM, a new hardware-assisted storage virtualization mechanism. The key idea is to implement (1) a storage virtualization layer on an FPGA card (FVM-engine) decoupled from the host resources and (2) a device-control method to have the card directly manage the physical storage devices. In this way, a server equipped with FVM-engine achieves high performance, scalability, and flexibility by saving the invaluable host-side resources and by adding the decoupled VM management-efficient FPGA cards as needed.

Acknowledgments

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-12. We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center (ISRC) at Seoul National University.

References

[1] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[2] Broadcom Stingray SmartNIC Adapters. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[3] DigiKey A-U50DD-P00G-ES3-G. https://www.digikey.com/en/products/detail/xilinx-inc/A-U50DD-P00G-ES3-G/10642492.

[4] DPDK. https://www.dpdk.org/.

[5] Flexible I/O Tester. https://github.com/axboe/fio.

[6] Google Cloud Computing Services. https://cloud.google.com/.

[7] Intel Optane SSD 900P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-900p-series.html.

[8] Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[9] Intel® Xeon® Gold 5118 Processor. https://ark.intel.com/content/www/us/en/ark/products/120473/intel-xeon-gold\-5118-processor-16-5m-cache-2-30-ghz.html.

[10] Intel® Xeon® Processor E5-4669 v4. https://ark.intel.com/content/www/us/en/ark/products/93805/intel-xeon-processor\-e5-4669-v4-55m-cache-2-20-ghz.html.

[11] Linux KVM. https://www.linux-kvm.org/.

[12] Microsoft Azure Cloud Computing Services. https://azure.microsoft.com/en-us/.

[13] NVIDIA Mellanox BlueField-2 DPU. https://www.mellanox.com/products/bluefield2-overview.

[14] NVM Express. https://nvmexpress.org/.

[15] QEMU. https://www.qemu.org/.

[16] RocksDB - A Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org/.

[17] Samsung PM1733 NVMe SSD. https://www.samsung.com/semiconductor/ssd/enterprise-ssd/MZWLJ3T8HBLS-00007/.

[18] Samsung Z-SSD. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[19] Single-Root Input/Output Virtualization. http://www.pcisig.com/specifications/.

[20] SPDK. https://spdk.io/.

[21] SPDK I/O Virtualization with Vhost-user. https://spdk.io/doc/vhost_processing.html.

[22] Super Micro Computer, SuperServer, 4029GP-TRT2. https://www.supermicro.com/en/products/system/4U/4029/SYS-4029GP-TRT2.cfm.

[23] Xilinx Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[24] Xilinx MicroBlaze Soft Processor Core. https://www.xilinx.com/products/design-tools/microblaze.html.

[25] Xilinx QDMA Subsystem for PCI Express. https://www.xilinx.com/products/intellectual-property/pcie-qdma.html.

[26] AMD I/O Virtualization Technology (IOMMU) Specification, Rev 1.26. 2009.

[27] Intel® Virtualization Technology for Directed I/O, Rev 1.3. 2011.

[28] Xilinx QDMA Subsystem for PCI Express v3.0. 2019.

[29] Xilinx UltraScale Architecture Memory Resources v1.11. 2020.

[30] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 10(3), 2006.

[31] Elena Agostini, Davide Rossetti, and Sreeram Potluri. Offloading communication control logic in gpu accelerated applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 248–257. IEEE, 2017.

[32] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. Dcs: a fast and scalable device-centric server architecture. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 559–571. IEEE, 2015.

[33] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011.

[34] Andrea Arcangeli. Micro-optimizing kvm vm-exits. In *KVM Forum*, 2019.

[35] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. Cloudcache: On-demand flash cache management for cloud computing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 355–369, 2016.

[36] Peter A Balinski, Sasikanth Eda, Ashwin M Joshi, John T Olson, and Sandeep R Patil. Dynamic i/o throttling in a storlet environment, March 10 2020. US Patent 10,585,596.

[37] Deepavali Bhagwat, Mahesh Patil, Michal Ostrowski, Murali Vilayannur, Woon Jung, and Chethan Kumar. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 287–300, 2015.

[38] Erwin Tam Raghu Ramakrishnan Brian F. Cooper, Adam Silberstein and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154. IEEE, 2010.

[39] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 329–344, 2017.

[40] Hoi Chan and Trieu Chieu. An approach to high availability for cloud servers with snapshot mechanism. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, pages 1–6, 2012.

[41] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

[42] NVM Express. NVM Express revision 1.3 specification. page 220, 2017.

[43] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, page 153–165, 2016.

[44] Jim Harris. Accelerating NVMe-oF* for VMs with the Storage Performance Development Kit. In *Flash Memory Summit*, 2017.

[45] Asias He. Virtio-blk Performance Improvement. In *KVM Forum*, 2012.

[46] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, 2017.

[47] Masaki Kimura. Better Utilization of Storage Features from KVM Guest via virtio-scsi. In *LinuxCon and CloudOpen North America*, 2013.

[48] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual remote i/o. *ACM SIGARCH Computer Architecture News*, 44(2):49–65, 2016.

[49] Dongup Kwon, Jaehyung Ahn, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim. Dcs-ctrl: a fast and flexible device-control mechanism for device-centric server architecture. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 491–504. IEEE, 2018.

[50] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[51] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. Splitx: Split guest/hypervisor execution on multi-core. In *WIOV*, 2011.

[52] Emmanuel S Levijarvi and Ognian S Mitzev. Private cloud replication and recovery, January 6 2015. US Patent 8,930,747.

[53] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[54] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. Pslo: Enforcing the xth percentile latency and throughput slos for consolidated vm storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–14, 2016.

[55] Anthony Liguori. The Nitro Project – Next Generation AWS Infrastructure. In *Hot Chips: A Symposium on High Performance Chips*, 2018.

[56] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[57] David Matlack. Kvm message passing performance. In *KVM Forum*, 2015.

[58] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX Annual technical conference, general track*, pages 391–394, 2005.

[59] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. Mdev-nvme: a nvme storage virtualization solution with mediated pass-through. In *2018 USENIX Annual Technical Conference (ATC 18)*, pages 665–676, 2018.

[60] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[61] Yossi Saad, Assaf Natanzon, and Yedidya Dotan. Securing data replication, backup and mobility in cloud storage, October 6 2015. US Patent 9,152,578.

[62] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE, 2008.

[63] Uma Somani, Kanika Lakhani, and Manish Mundra. Implementing digital signature with rsa encryption algorithm to enhance the data security of cloud in cloud computing. In *2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010)*, pages 211–216. IEEE, 2010.

[64] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–131, 2020.

[65] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. Srcmap: Energy proportional storage using dynamic consolidation. In *FAST*, volume 10, pages 267–280, 2010.

[66] Carl Waldspurger and Mendel Rosenblum. I/O Virtualization. *Communications of the ACM*, 55(1):66–73, 2012.

[67] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[68] Jiwei Xu, Wenbo Zhang, Shiyang Ye, Jun Wei, and Tao Huang. A lightweight virtual machine image deduplication backup approach in cloud environment. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 503–508. IEEE, 2014.

[69] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. Spdk vhost-nvme: Accelerating i/os in virtual machines on nvme ssds via user space vhost target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76. IEEE, 2018.

[70] Lei Yu, Chuliang Weng, Minglu Li, and Yuan Luo. Snpdisk: an efficient para-virtualization snapshot mechanism for virtual disks in private clouds. *IEEE Network*, 25(4):20–26, 2011.

[71] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.

[72] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *2010 IEEE international conference on cluster computing*, pages 88–96. IEEE, 2010.



hXDP: Efficient Software Packet Processing on FPGA NICs

Marco Spaziani Brunella[1,3], Giacomo Belocchi[1,3], Marco Bonola[1,2], Salvatore Pontarelli[1], Giuseppe Siracusano[4], Giuseppe Bianchi[3], Aniello Cammarano[2,3], Alessandro Palumbo[2,3], Luca Petrucci[2,3] and Roberto Bifulco[4]

[1]Axbryd, [2]CNIT, [3]University of Rome Tor Vergata, [4]NEC Laboratories Europe

Abstract

FPGA accelerators on the NIC enable the offloading of expensive packet processing tasks from the CPU. However, FPGAs have limited resources that may need to be shared among diverse applications, and programming them is difficult.

We present a solution to run Linux's eXpress Data Path programs written in eBPF on FPGAs, using only a fraction of the available hardware resources while matching the performance of high-end CPUs. The iterative execution model of eBPF is not a good fit for FPGA accelerators. Nonetheless, we show that many of the instructions of an eBPF program can be compressed, parallelized or completely removed, when targeting a purpose-built FPGA executor, thereby significantly improving performance. We leverage that to design hXDP, which includes (i) an optimizing-compiler that parallelizes and translates eBPF bytecode to an extended eBPF Instruction-set Architecture defined by us; a (ii) soft-CPU to execute such instructions on FPGA; and (iii) an FPGA-based infrastructure to provide XDP's maps and helper functions as defined within the Linux kernel.

We implement hXDP on an FPGA NIC and evaluate it running real-world unmodified eBPF programs. Our implementation is clocked at 156.25MHz, uses about 15% of the FPGA resources, and can run dynamically loaded programs. Despite these modest requirements, it achieves the packet processing throughput of a high-end CPU core and provides a 10x lower packet forwarding latency.

1 Introduction

FPGA-based NICs have recently emerged as a valid option to offload CPUs from packet processing tasks, due to their good performance and re-programmability. Compared to other NIC-based accelerators, such as network processing ASICs [8] or many-core System-on-Chip SmartNICs [40], FPGA NICs provide the additional benefit of supporting diverse accelerators for a wider set of applications [42], thanks to their embedded hardware re-programmability. Notably, Microsoft has been advocating for the introduction of FPGA NICs, because of their ability to use the FPGAs also for tasks such as machine learning [13, 14]. FPGA NICs play another important role in 5G telecommunication networks, where they are used for the acceleration of radio access network functions [11, 28, 39, 58]. In these deployments, the FPGAs could host multiple functions to provide higher levels of infrastructure consolidation, since physical space availability may be limited. For instance, this is the case in smart cities [55], 5G local deployments, e.g., in factories [44, 47], and for edge computing in general [6, 30]. Nonetheless, programming FPGAs is difficult, often requiring the establishment of a dedicated team composed of hardware specialists [18], which interacts with software and operating system developers to integrate the offloading solution with the system. Furthermore, previous work that simplifies network functions programming on FPGAs assumes that a large share of the FPGA is dedicated to packet processing [1, 45, 56], reducing the ability to share the FPGA with other accelerators.

In this paper, our goal is to provide a more general and easy-to-use solution to program packet processing on FPGA NICs, using little FPGA resources, while seamlessly integrating with existing operating systems. We build towards this goal by presenting `hXDP`, a set of technologies that enables the efficient execution of the Linux's `eXpress Data Path` (XDP) [27] on FPGA. XDP leverages the eBPF technology to provide secure programmable packet processing within the Linux kernel, and it is widely used by the Linux's community in productive environments. hXDP provides full XDP support, allowing users to dynamically load and run their unmodified XDP programs on the FPGA.

The eBPF technology is originally designed for sequential execution on a high-performance RISC-like register machine, which makes it challenging to run XDP programs effectively on FPGA. That is, eBPF is designed for server CPUs with high clock frequency and the ability to execute many of the sequential eBPF instructions per second. Instead, FPGAs favor a widely parallel execution model with clock frequencies that are 5-10x lower than those of high-end CPUs. As such, a straightforward implementation of the eBPF iterative exe-

cution model on FPGA is likely to provide low packet forwarding performance. Furthermore, the hXDP design should implement arbitrary XDP programs while using little hardware resources, in order to keep FPGA's resources free for other accelerators.

We address the challenge performing a detailed analysis of the eBPF Instruction Set Architecture (ISA) and of the existing XDP programs, to reveal and take advantage of opportunities for optimization. First, we identify eBPF instructions that can be safely removed, when not running in the Linux kernel context. For instance, we remove data boundary checks and variable zero-ing instructions by providing targeted hardware support. Second, we define extensions to the eBPF ISA to introduce 3-operand instructions, new 6B load/store instructions and a new parametrized program exit instruction. Finally, we leverage eBPF instruction-level parallelism, performing a static analysis of the programs at compile time, which allows us to execute several eBPF instructions in parallel. We design hXDP to implement these optimizations, and to take full advantage of the on-NIC execution environment, e.g., avoiding unnecessary PCIe transfers. Our design includes: (i) a compiler to translate XDP programs' bytecode to the extended hXDP ISA; (ii) a self-contained FPGA IP Core module that implements the extended ISA alongside several other low-level optimizations; (iii) and the toolchain required to dynamically load and interact with XDP programs running on the FPGA NIC.

To evaluate hXDP we provide an open source implementation for the NetFPGA [60]. We test our implementation using the XDP example programs provided by the Linux source code, and using two real-world applications: a simple stateful firewall; and Facebook's Katran load balancer. hXDP can match the packet forwarding throughput of a multi-GHz server CPU core, while providing a much lower forwarding latency. This is achieved despite the low clock frequency of our prototype (156MHz) and using less than 15% of the FPGA resources. In summary, we contribute:

- the design of hXDP including: the hardware design; the companion compiler; and the software toolchain;
- the implementation of a hXDP IP core for the NetFPGA
- a comprehensive evaluation of hXDP when running real-world use cases, comparing it with an x86 Linux server.
- a microbenchmark-based comparison of the hXDP implementation with a Netronome NFP4000 SmartNIC, which provides partial eBPF offloading support.

2 Concept and overview

In this section we discuss hXDP goals, scope and requirements, we provide background information about XDP, and finally we present an overview of the hXDP design.

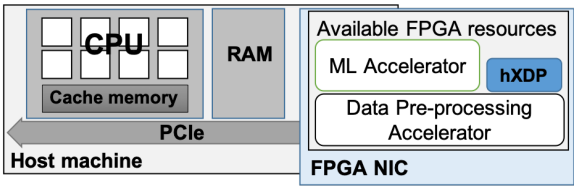


Figure 1: The hXDP concept. hXDP provides an easy-to-use network accelerator that shares the FPGA NIC resources with other application-specific accelerators.

2.1 Goals and Requirements

Goals Our main goal is to provide the ability to run XDP programs efficiently on FPGA NICs, while using little FPGA's hardware resources (See Figure 1).

A little use of the FPGA resources is especially important, since it enables extra consolidation by packing different application-specific accelerators on the same FPGA.

The choice of supporting XDP is instead motivated by a twofold benefit brought by the technology: it readily enables NIC offloading for already deployed XDP programs; it provides an on-NIC programming model that is already familiar to a large community of Linux programmers. Enabling such a wider access to the technology is important since many of the mentioned edge deployments are not necessarily handled by hyperscale companies. Thus, the companies developing and deploying applications may not have resources to invest in highly specialized and diverse professional teams of developers, while still needing some level of customization to achieve challenging service quality and performance levels. In this sense, hXDP provides a familiar programming model that does not require developers to learn new programming paradigms, such as those introduced by devices that support P4 [7] or FlowBlaze [45].

Non-Goals Unlike previous work targeting FPGA NICs [1, 45, 56], hXDP does not assume the FPGA to be dedicated to network processing tasks. Because of that, hXDP adopts an iterative processing model, which is in stark contrast with the pipelined processing model supported by previous work. The iterative model requires a fixed amount of resources, no matter the complexity of the program being implemented. Instead, in the pipeline model the resource requirement is dependent on the implemented program complexity, since programs are effectively "unrolled" in the FPGA. In fact, hXDP provides dynamic runtime loading of XDP programs, whereas solutions like P4->NetFPGA [56] or FlowBlaze need to often load a new FPGA bitstream when changing application. As such, hXDP is not designed to be faster at processing packets than those designs. Instead, hXDP aims at freeing precious CPU resources, which can then be dedicated to workloads that cannot run elsewhere, while providing similar or better performance than the CPU.

Likewise, hXDP cannot be directly compared to SmartNICs dedicated to network processing. Such NICs' resources

are largely, often exclusively, devoted to network packet processing. Instead, hXDP leverages only a fraction of an FPGA resources to add packet processing with good performance, alongside other application-specific accelerators, which share the same chip's resources.

Finally, hXDP is not providing a transparent offloading solution.[1] While the programming model and the support for XDP are unchanged compared to the Linux implementation, programmers should be aware of which device runs their XDP programs. This is akin to programming for NUMA systems, in which accessing given memory areas may incur additional overheads.

Requirements Given the above discussion, we can derive three high-level requirements for hXDP:

1. it should execute unmodified compiled XDP programs, and support the XDP frameworks' toolchain, e.g., dynamic program loading and userspace access to maps;
2. it should provide packet processing performance at least comparable to that of a high-end CPU core;
3. it should require a small amount of the FPGA's hardware resources.

Before presenting a more detailed description of the hXDP concept, we now give a brief background about XDP.

2.2 XDP Primer

XDP allows programmers to inject programs at the NIC driver level, so that such programs are executed before a network packet is passed to the Linux's network stack. This provides an opportunity to perform custom packet processing at a very early stage of the packet handling, limiting overheads and thus providing high-performance. At the same time, XDP allows programmers to leverage the Linux's kernel, e.g., selecting a subset of packets that should be processed by its network stack, which helps with compatibility and ease of development. XDP is part of the Linux kernel since release 4.18, and it is widely used in production environments [4, 17, 54]. In most of these use cases, e.g., load balancing [17] and packet filtering [4], a majority of the received network packets is processed entirely within XDP. The production deployments of XDP have also pushed developers to optimize and minimize the XDP overheads, which now appear to be mainly related to the Linux driver model, as thoroughly discussed in [27].

XDP programs are based on the Linux's eBPF technology. eBPF provides an in-kernel virtual machine for the sandboxed execution of small programs within the kernel context. An overview of the eBPF architecture and workflow is provided in Figure 2. In its current version, the eBPF virtual machine has 11 64b registers: $r0$ holds the return value from in-kernel functions and programs, $r1 - r5$ are used to store arguments that are passed to in-kernel functions, $r6 - r9$ are registers

[1] Here, previous complementary work may be applied to help the automatic offloading of network processing tasks [43].

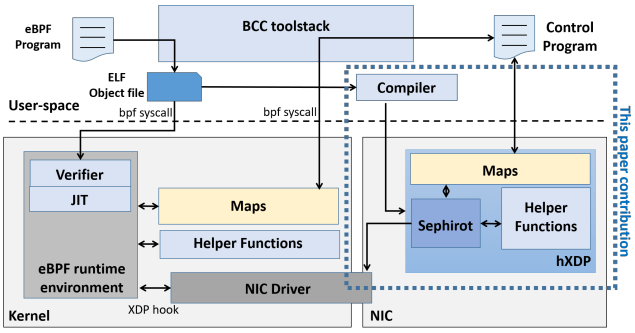


Figure 2: An overview of the XDP workflow and architecture, including the contribution of this paper.

that are preserved during function calls and $r10$ stores the frame pointer to access the stack. The eBPF virtual machine has a well-defined ISA composed of more than 100 fixed length instructions (64b). The instructions give access to different functional units, such as ALU32, ALU64, branch and memory. Programmers usually write an eBPF program using the C language with some restrictions, which simplify the static verification of the program. Examples of restrictions include forbidden unbounded cycles, limited stack size, lack of dynamic memory allocation, etc.

To overcome some of these limitations, eBPF programs can use `helper functions` that implement some common operations, such as checksum computations, and provide access to protected operations, e.g., reading certain kernel memory areas. eBPF programs can also access kernel memory areas called `maps`, i.e., kernel memory locations that essentially resemble tables. Maps are declared and configured at compile time to implement different data structures, specifying the type, size and an ID. For instance, eBPF programs can use maps to implement arrays and hash tables. An eBPF program can interact with map's locations by means of pointer deference, for un-structured data access, or by invoking specific helper functions for structured data access, e.g., a lookup on a map configured as a hash table. Maps are especially important since they are the only mean to keep state across program executions, and to share information with other eBPF programs and with programs running in user space. In fact, a map can be accessed using its ID by any other running eBPF program and by the control application running in user space. User space programs can load eBPF programs and read/write maps either using the `libbf` library or frontends such as the BCC toolstack.

XDP programs are compiled using LLVM or GCC, and the generated ELF object file is loaded trough the `bpf` syscall, specifying the `XDP hook`. Before the actual loading of a program, the kernel verifier checks if it is safe, then the program is attached to the hook, at the network driver level. Whenever the network driver receives a packet, it triggers the execution of the registered programs, which starts from a clean context.

2.3 Challenges

To grasp an intuitive understanding of the design challenge involved in supporting XDP on FPGA, we now consider the example of an XDP program that implements a simple stateful firewall for checking the establishment of bi-directional TCP or UDP flows, and to drop flows initiated from an external location. We will use this function as a running example throughout the paper, since despite its simplicity, it is a realistic and widely deployed function.

The simple firewall first performs a parsing of the Ethernet, IP and Transport protocol headers to extract the flow's 5-tuple (IP addresses, port numbers, protocol). Then, depending on the input port of the packet (i.e., external or internal) it either looks up an entry in a hashmap, or creates it. The hashmap key is created using an absolute ordering of the 5 tuple values, so that the two directions of the flow will map to the same hash. Finally, the function forwards the packet if the input port is internal or if the hashmap lookup retrieved an entry, otherwise the packet is dropped. A C program describing this simple firewall function is compiled to 71 eBPF instructions.

We can build a rough idea of the potential best-case speed of this function running on an FPGA-based eBPF executor, assuming that each eBPF instruction requires 1 clock cycle to be executed, that clock cycles are not spent for any other operation, and that the FPGA has a 156MHz clock rate, which is common in FPGA NICs [60]. In such a case, a naive FPGA implementation that implements the sequential eBPF executor would provide a maximum throughput of 2.8 Million packets per second (Mpps).[2] Notice that this is a very optimistic upper-bound performance, which does not take into account other, often unavoidable, potential sources of overhead, such as memory access, queue management, etc. For comparison, when running on a single core of a high-end server CPU clocked at 3.7GHz, and including also operating system overhead and the PCIe transfer costs, the XDP simple firewall program achieves a throughput of 7.4 Million packets per second (Mpps). [3] Since it is often undesired or not possible to increase the FPGA clock rate, e.g., due to power constraints, in the lack of other solutions the FPGA-based executor would be 2-3x slower than the CPU core.

Furthermore, existing solutions to speed-up sequential code execution, e.g., superscalar architectures, are too expensive in terms of hardware resources to be adopted in this case. In fact, in a superscalar architecture the speed-up is achieved leveraging instruction-level parallelism at runtime. However, the complexity of the hardware required to do so grows exponentially with the number of instructions being checked for parallel execution. This rules out re-using general purpose soft-core designs, such as those based on RISC-V [22, 25].

[2] I.e., the FPGA can run 156M instructions per second, which divided by the 55 instructions of the program's expected execution path gives a 2.8M program executions per second. Here, notice that the execution path comprises less instructions that the overall program, since not all the program's instructions are executed at runtime due to if-statements.

[3] Intel Xeon E5-1630v3, Linux kernel v.5.6.4.

2.4 hXDP Overview

hXDP addresses the outlined challenge by taking a software-hardware co-design approach. In particular, hXDP provides both a compiler and the corresponding hardware module. The compiler takes advantage of eBPF ISA optimization opportunities, leveraging hXDP's hardware module features that are introduced to simplify the exploitation of such opportunities. Effectively, we design a new ISA that extends the eBPF ISA, specifically targeting the execution of XDP programs.

The compiler optimizations perform transformations at the eBPF instruction level: remove unnecessary instructions; replace instructions with newly defined more concise instructions; and parallelize instructions execution. All the optimizations are performed at compile-time, moving most of the complexity to the software compiler, thereby reducing the target hardware complexity. We describe the optimizations and the compiler in Section 3. Accordingly, the hXDP hardware module implements an infrastructure to run up to 4 instructions in parallel, implementing a Very Long Instruction Word (VLIW) soft-processor. The VLIW soft-processor does not provide any runtime program optimization, e.g., branch prediction, instruction re-ordering, etc. We rely entirely on the compiler to optimize XDP programs for high-performance execution, thereby freeing the hardware module of complex mechanisms that would use more hardware resources. We describe the hXDP hardware design in Section 4.

Ultimately, the hXDP hardware component is deployed as a self-contained IP core module to the FPGA. The module can be interfaced with other processing modules if needed, or just placed as a bump-in-the-wire between the NIC's port and its PCIe driver towards the host system. The hXDP software toolchain, which includes the compiler, provides all the machinery to use hXDP within a Linux operating system.

From a programmer perspective, a compiled eBPF program could be therefore interchangeably executed in-kernel or on the FPGA, as shown in Figure 2.[4]

3 hXDP Compiler

In this section we describe the hXDP instruction-level optimizations, and the compiler design to implement them.

3.1 Instructions reduction

The eBPF technology is designed to enable execution within the Linux kernel, for which it requires programs to include a number of extra instructions, which are then checked by the kernel's verifier. When targeting a dedicated eBPF executor implemented on FPGA, most such instructions could be

[4] The choice of where to run an XDP program should be explicitly taken by the user, or by an automated control and orchestration system, if available.

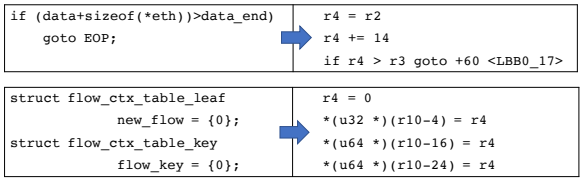


Figure 3: Examples of instructions removed by hXDP

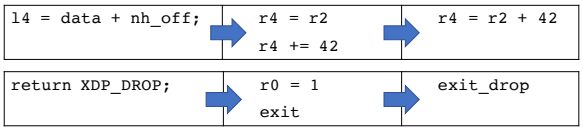


Figure 4: Examples of hXDP ISA extensions

safely removed, or they can be replaced by cheaper embedded hardware checks. Two relevant examples are instructions for memory boundary checks and memory zero-ing.

Boundary checks are required by the eBPF verifier to ensure that programs only read valid memory locations, whenever a pointer operation is involved. For instance, this is relevant for accessing the socket buffer containing the packet data, during parsing. Here, a required check is to verify that the packet is large enough to host the expected packet header. As shown in Figure 3, a single check like this may cost 3 instructions, and it is likely that such checks are repeated multiple times. In the simple firewall case, for instance, there are three such checks for the Ethernet, IP and L4 headers. In hXDP we can safely remove these instructions, implementing the check directly in hardware.

Zero-ing is the process of setting a newly created variable to zero, and it is a common operation performed by programmers both for safety and for ensuring correct execution of their programs. A dedicated FPGA executor can provide hard guarantees that all relevant memory areas are zero-ed at program start, therefore making the explicit zero-ing of variables during initialization redundant. In the simple firewall function zero-ing requires 4 instructions, as shown in Figure 3.

3.2 ISA extension

To effectively reduce the number of instructions we define an ISA that enables a more concise description of the program. Here, there are two factors at play to our advantage. First, we can extend the ISA without accounting for constraints related to the need to support efficient Just-In-Time compilation. Second, our eBPF programs are part of XDP applications, and as such we can expect packet processing as the main program task. Leveraging these two facts we define a new ISA that changes in three main ways the original eBPF ISA.

Operands number. The first significant change has to deal with the inclusion of three-operand operations, in place of eBPF's two-operand ones. Here, we believe that the eBPF's ISA selection of two-operand operations was mainly dictated by the assumption that an x86 ISA would be the final compilation target. Instead, using three-operand instructions allows us to implement an operation that would normally need two instructions with just a single instruction, as shown in Figure 4.

Load/store size. The eBPF ISA includes byte-aligned memory load/store operations, with sizes of 1B, 2B, 4B and 8B. While these instructions are effective for most cases, we noticed that during packet processing the use of 6B load/store can reduce the number of instructions in common cases. In fact, 6B is the size of an Ethernet MAC address, which is a commonly accessed field both to check the packet destination or to set a new one. Extending the eBPF ISA with 6B load/store instructions often halves the required instructions.

Parametrized exit. The end of an eBPF program is marked by the exit instruction. In XDP, programs set the $r0$ to a value corresponding to the desired forwarding action (e.g., DROP, TX, etc), then, when a program exits the framework checks the $r0$ register to finally perform the forwarding action (see listing 4). While this extension of the ISA only saves one (runtime) instruction per program, as we will see in Section 4, it will also enable more significant hardware optimizations.

3.3 Instruction Parallelism

Finally, we explore the opportunity to perform parallel processing of an eBPF program's instructions. Here, it is important to notice that high-end *superscalar* CPUs are usually capable to execute multiple instructions in parallel, using a number of complex mechanisms such as speculative execution or out-of-order execution. However, on FPGAs the introduction of such mechanisms could incur significant hardware resources overheads. Therefore, we perform only a static analysis of the instruction-level parallelism of eBPF programs.

To determine if two or more instructions can be parallelized, the three Bernstein conditions have to be checked [3]. Simplifying the discussion to the case of two instructions P_1, P_2:

$$I_1 \cap O_2 = \emptyset; O_1 \cap I_2 = \emptyset; O_2 \cap O_1 = \emptyset; \quad (1)$$

Where I_1, I_2 are the instructions' input sets (e.g. source operands and memory locations) and O_1, O_2 are their output sets. The first two conditions imply that if any of the two instructions depends on the results of the computation of the other, those two instructions cannot be executed in parallel. The last condition implies that if both instructions are storing the results on the same location, again they cannot be parallelized. Verifying the Bernstein conditions and parallelizing instructions requires the design of a suitable compiler, which we describe next.

3.4 Compiler design

We design a custom compiler to implement the optimizations outlined in this section and to transform XDP programs into

a schedule of parallel instructions that can run with hXDP. The schedule can be visualized as a virtually infinite set of rows, each with multiple available spots, which need to be filled with instructions. The number of spots corresponds to the number of execution lanes of the target executor. The final objective of the compiler is to fit the given XDP program's instructions in the smallest number of rows. To do so, the compiler performs five steps.

***Control Flow Graph* construction** First, the compiler performs a forward scan of the eBPF bytecode to identify the program's *basic blocks*, i.e., sequences of instructions that are always executed together. The compiler identifies the first and last instructions of a block, and the control flow between blocks, by looking at branching instructions and jump destinations. With this information it can finally build the *Control Flow Graph* (CFG), which represents the basic blocks as nodes and the control flow as directed edges connecting them.

***Peephole* optimizations** Second, for each basic block the compiler performs the removal of unnecessary instructions (cf. Section 3.1), and the substitution of groups of eBPF instructions with an equivalent instruction of our extended ISA (cf. Section 3.2).

Data Flow dependencies Third, the compiler discovers *Data Flow dependencies*. This is done by implementing an iterative algorithm to analyze the CFG. The algorithm analyzes each block, building a data structure containing the block's input, output, defined, and used symbols. Here, a symbol is any distinct data value defined (and used) by the program. Once each block has its associated set of symbols, the compiler can use the CFG to compute data flow dependencies between instructions. This information is captured in per-instruction *data dependency graphs* (DDG).

Instruction scheduling Fourth, the compiler uses the CFG and the learned DDGs to define an instruction schedule that meets the first two Bernstein conditions. Here, the compiler takes as input the maximum number of parallel instructions the target hardware can execute, and potential hardware constraints it needs to account for. For example, as we will see in Section 4, the hXDP executor has 4 parallel execution lanes, but helper function calls cannot be parallelized.

To build the instructions schedule, the compiler considers one basic block at a time, in their original order in the CFG. For each block, the compiler assigns the instructions to the current schedule's row, starting from the first instruction in the block and then searching for any other *enabled* instruction. An instruction is enabled for a given row when its data dependencies are met, and when the potential hardware constraints are respected. E.g., an instruction that calls a helper function is not enabled for a row that contains another such instruction. If the compiler cannot find any enabled instruction for the current row, it creates a new row. The algorithm continues until all the block's instructions are assigned to a row.

At this point, the compiler uses the CFG to identify potential candidate blocks whose instructions may be added to the schedule being built for the current block. That is, such block's instructions may be used to fill gaps in the current schedule's rows. The compiler considers as candidate blocks the current block's *control equivalent blocks*. I.e, those blocks that are surely going to be executed if the current block is executed. Instructions from such blocks are checked and, if enabled, they are added to the currently existing schedule's rows. This allows the compiler to move in the current block's schedule also a series of branching instructions that are immediately following the current block, enabling a *parallel branching* optimization in hardware (cf. Section 4.2).

When the current block's and its candidate blocks' enabled instructions are all assigned, the algorithm moves to the next block with instructions not yet scheduled, re-applying the above steps. The algorithm terminates once all the instructions have been assigned to the schedule.

Physical register assignment Finally, in the last step the compiler assigns physical registers to the program's symbols. First, the compilers assigns registers that have a precise semantic, such as `r0` for the exit code, `r1`-`r5` for helper function argument passing, and `r10` for the frame pointer. After these fixed assignment, the compiler checks if for every row also the third Bernstein condition is met, otherwise it renames the registers of one of the conflicting instructions, and propagates the renaming on the following dependant instructions.

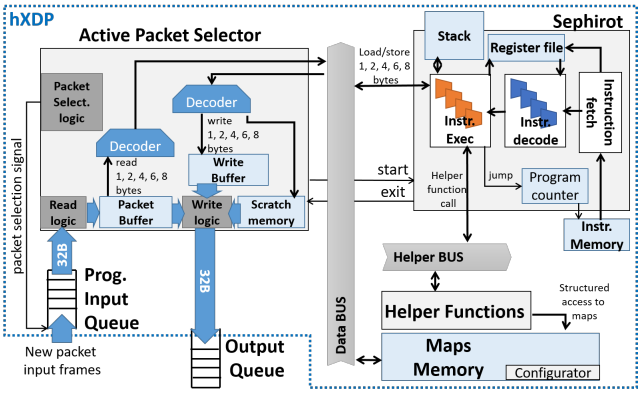


Figure 5: The logic architecture of the hXDP hardware design.

4 Hardware Module

We design hXDP as an independent IP core, which can be added to a larger FPGA design as needed. Our IP core comprises the elements to execute all the XDP functional blocks on the NIC, including `helper functions` and `maps`. This enables the execution of a program entirely on the FPGA NIC and therefore it avoids as much as possible PCIe transfers.

4.1 Architecture and components

The hXDP hardware design includes five components (see Figure 5): the Programmable Input Queue (`PIQ`); the Ac-

tive Packet Selector (APS); the Sephirot processing core; the Helper Functions Module (HF); and the Memory Maps Module (MM). All the modules work in the same clock frequency domain. Incoming data is received by the PIQ. The APS reads a new packet from the PIQ into its internal packet buffer. In doing so, the APS provides a byte-aligned access to the packet data through a *data bus*, which Sephirot uses to selectively read/write the packet content. When the APS makes a packet available to the Sephirot core, the execution of a loaded eBPF program starts. Instructions are entirely executed within Sephirot, using 4 parallel execution lanes, unless they call a helper function or read/write to maps. In such cases, the corresponding modules are accessed using the *helper bus* and the *data bus*, respectively. We detail each components next.

4.1.1 Programmable Input queue

When a packet is received, it enters the Programmable Input Queue (PIQ), which works as an interface with the NIC input bus. Thus, a packet is usually received divided into *frames*, received at each clock cycle. The PIQ holds the packet's frames maintaining a *head frame pointer*. The frames of a given packet can be therefore read from the queue independently from the reception order.

4.1.2 Active Packet Selector

The APS implements a finite-state machine to handle the transfer of a selected packet's frames from the PIQ to an APS' internal buffer.[5] The internal buffer is large enough to hold a full-sized packet.

While the packet is stored divided in frames, the APS provides a byte-aligned read/write access to the data, as required by the eBPF ISA. I.e., the APS implements an eBPF program's *packet buffer*, and builds the hardware-equivalent of the xdp_md struct that is passed as argument to XDP programs. Sephirot accesses such data structure using the main hXDP's data bus. Since Sephirot has four parallel execution lanes, the APS provides four parallel read/write memory accesses through the data bus.

Storing the packet data in frames simplifies the buffer implementation. Nonetheless, this also makes the writing of specific bytes in the packet more complex. In particular, since only a frame-size number of bytes can be written to the buffer, the writing of single bytes would need to first read the entire frame, apply the single-byte modification, and then re-write to the buffer the entire modified frame. This is a complex operation, which would impact the maximum achievable clock rate if implemented in this way, or it would alternatively require multiple clock cycles to be completed. We instead use a *difference buffer* to handle writes, trading off some memory space for hardware complexity. That is, modifications to the packet data are stored in a difference buffer that is byte addressed. As we will see next, the difference buffer allows us to separate the reading of certain packet data, which can be pre-fetched by Sephirot during the decoding of an instruction, from the actual writing of new data in the packet, which can be performed during packet emission. In fact, the APS contains also a scratch memory to handle modifications to the packet that are applied before the current packet head. This is usually required by applications that use the bpf_adjust_head helper.

The scratch memory, the difference buffer, and the packet buffer are combined when Sephirot performs a read of the packet data, and at the end of the program execution, during packet emission. Packet emission is the process of moving the modified packet data to the output queue. The entire process is handled by a dedicated finite-state machine, which is started by Sephirot when an *exit* instruction is executed. The emission of a packet happens in parallel with the reading of the next packet.

4.1.3 Sephirot

Sephirot is a VLIW processor with 4 parallel lanes that execute eBPF instructions. Sephirot is designed as a pipeline of four stages: instruction fetch (IF); instruction decode (ID); instruction execute (IE); and commit. A program is stored in a dedicated instruction memory, from which Sephirot fetches the instructions in order. The processor has another dedicated memory area to implement the program's stack, which is 512B in size, and 11 64b registers stored in the register file. These memory and register locations match one-to-one the eBPF virtual machine specification. Sephirot begins execution when the APS has a new packet ready for processing, and it gives the processor *start* signal.

On processor start (IF stage) a VLIW instruction is read and the 4 extended eBPF instructions that compose it are statically assigned to their respective execution lanes. In this stage, the operands of the instructions are pre-fetched from the register file. The remaining 3 pipeline stages are performed in parallel by the four execution lanes. During ID, memory locations are pre-fetched, if any of the eBPF instructions is a *load*, while at the IE stage the relevant sub-unit are activated, using the relevant pre-fetched values. The sub-units are the Arithmetic and Logic Unit (ALU), the Memory Access Unit and the Control Unit. The ALU implements all the operations described by the eBPF ISA, with the notable difference that it is capable of performing operations on three operands. The memory access unit abstracts the access to the different memory areas, i.e., the stack, the packet data stored in the APS, and the maps memory. The control unit provides the logic to modify the program counter, e.g., to perform a *jump*, and to invoke helper functions. Finally, during the commit stage the results of the IE phase are stored back to the register file, or to one of the

[5] The policy for selecting a given packet from the PIQ is by default FIFO, although this can be changed to implement more complex mechanisms.

memory areas. Sephirot terminates execution when it finds an exit instruction, in which case it signals to the APS the packet forwarding decision.

4.1.4 Helper Functions

hXDP implements the XDP helper functions in a dedicated sub-module. We decided to provide a dedicated hardware implementation for the helper functions since their definition is rather static, and it changes seldom when new versions of the XDP technology are released. This also allows us to leverage at full the FPGA hardware parallelism to implement some more expensive functions, such as checksum computations. In terms of interface, the helper function sub-module offers the same interface provided by eBPF, i.e., helper functions arguments are read from registers r1-r5, and the return value is provided in r0. All values are exchanged using the dedicated helper data bus. Here, it is worth noticing that there is a single helper functions sub-module, and as such only one instruction per cycle can invoke a helper function.[6] Among the helper functions there are the map lookup functions, which are used to implement hashmap and other data structures on top of the maps memory. Because of that, the helper functions sub-module has a direct access to the maps module.

4.1.5 Maps

The maps subsystem main function is to decode memory addresses, i.e., map id and row, to access the corresponding map's memory location. Here, one complication is that eBPF maps can be freely specified by a program, which defines the map's type and size for as many maps as needed. To replicate this feature in the hardware, the maps subsystem implements a *configurator* which is instructed at program's load time. In fact, all the maps share the same FPGA memory area, which is then shaped by the configurator according to the maps section of the eBPF program, which (virtually) creates the appropriate number of maps with their row sizes, width and hash functions, e.g., for implementing hashmaps.

Since in eBPF single maps entries can be accessed directly using their address, the maps subsystem is connected via the data bus to Sephirot, in addition to the direct connection to the helper function sub-module, which is instead used for structured map access. To enable parallel access to the Sephirot's execution lanes, like in the case of the APS, the maps modules provides up to 4 read/write parallel accesses.

4.2 Pipeline Optimizations

Early processor start The packet content is transferred one frame per clock cycle from the PIQ to the APS. Starting program execution without waiting the full transfer of the packet may trigger the reading of parts of it that are not yet transferred. However, handling such an exception requires only little additional logic to pause the Sephirot pipeline, when the exception happens. In practice, XDP programs usually start reading the beginning of a packet, in fact in our tests we never experienced a case in which we had to pause Sephirot. This provides significant benefits with packets of larger sizes, effectively masking the Sephirot execution time.

```
switch (ip->protocol) {
      case IPPROTO_TCP: goto l4;
      case IPPROTO_UDP: goto l4;
      default:          goto EOP;}
```

```
if r1 == 17 goto +1 <LBB0_5>
if r1 != 6 goto +48 <LBB0_17>
```

Figure 6: Example of a switch statement

Program state self-reset As we have seen in Section 3, eBPF programs may perform zero-ing of the variables they are going to use. We provide automatic reset of the stack and of the registers at program initialization. This is an inexpensive feature in hardware, which improves security [15] and allows us to remove any such zero-ing instruction from the program.

Data hazards reduction One of the issues of pipelined execution is that two instructions executed back-to-back may cause a race condition. If the first instruction produces a result needed by the second one, the value read by the second instruction will be stale, because of the operand/memory pretecthing performed by Sephirot. Stalling the pipeline would avoid such race conditions at the cost of performance. Instead, we perform result forwarding on a per-lane basis. This allows the scheduling back-to-back of instructions on a single lane, even if the result of the first instruction is needed by the second one.[7] The compiler is in charge of checking such cases and ensure that the instructions that have such dependancies are always scheduled on the same lane.

Parallel branching The presence of branch instructions may cause performance problems with architectures that lack branch prediction, speculative and out of order execution. In the case of Sephirot, this forces a serialization of the branch instructions. However, in XDP programs there are often series of branches in close sequence, especially during header parsing (see Figure 6). We enabled the parallel execution of such branches, establishing a priority ordering of the Sephirot's lanes. That is, all the branch instructions are executed in parallel by the VLIW's lanes. If more than one branch is taken, the highest priority one is selected to update the program counter. The compiler takes that into account when scheduling instructions, ordering the branch instructions accordingly.[8]

Early processor exit The processor stops when an exit instruction is executed. The exit instruction is recognized during the IF phase, which allows us to stop the processor pipeline early, and save the 3 remaining clock cycles. This optimization

[6] Adding more sub-modules would not be sufficient to improve parallelism in this case, since we would need to also define additional registers to hold arguments/return values and include register renaming schemes. Adding sub-modules proved to be not helpful for most use cases.

[7] We empirically tested that a more complex intra-lane result forwarding does not provide measurable benefits.

[8] This applies equally to a sequence of if...else or goto statements.

Component	Logic	Registers	BRAM
PIQ	215, 0.05%	58, <0.01%	6.5, 0.44%
APS	9k, 2.09%	10k, 1.24%	4, 0.27%
Sephirot	27k, 6.35%	4k, 0,51%	-
Instr Mem	-	-	7.7, 0.51%
Stack	1k, 0.24%	136, 0.02%	16, 1,09%
HF Subsystem	339, 0.08%	150, 0.02%	-
Maps Subsystem	5.8k, 1.35%	2.5k, 0.3%	16, 1.09%
total	42k, 9.91%	18k, 2.09%	50, 3.40%
w/ reference NIC	80k, 18.53%	63k, 7.3%	214, 14,63%

Table 1: NetFPGA resources usage breakdown, each row reports actual number and percentage of the FPGA total resources (#, % tot). hXDP requires about 15% of the FPGA resources in terms of Slice Logic and Registers.

improves also the performance gain obtained by extending the ISA with parametrized exit instructions, as described in Section 3. In fact, XDP programs usually perform a move of a value to $r0$, to define the forwarding action, before calling an exit. Setting a value to a register always needs to traverse the entire `Sephirot` pipeline. Instead, with a parametrized exit we remove the need to assign a value to $r0$, since the value is embedded in a newly defined exit instruction.

4.3 Implementation

We prototyped hXDP using the NetFPGA [60], a board embedding 4 10Gb ports and a Xilinx Virtex7 FPGA. The hXDP implementation uses a frame size of 32B and is clocked at 156.25MHz. Both settings come from the standard configuration of the NetFPGA reference NIC design.

The hXDP FPGA IP core takes 9.91% of the FPGA logic resources, 2.09% of the register resources and 3.4% of the FPGA's available BRAM. The considered BRAM memory does not account for the variable amount of memory required to implement maps. A per-component breakdown of the required resources is presented in Table 1, where for reference we show also the resources needed to implement a map with 64 rows of 64B each. As expected, the APS and `Sephirot` are the components that need more logic resources, since they are the most complex ones. Interestingly, even somewhat complex helper functions, e.g., a helper function to implement a hashmap lookup (`HF Map Access`), have just a minor contribution in terms of required logic, which confirms that including them in the hardware design comes at little cost while providing good performance benefits, as we will see in Section 5. When including the NetFPGA's reference NIC design, i.e., to build a fully functional FPGA-based NIC, the overall occupation of resources grows to 18.53%, 7.3% and 14.63% for logic, registers and BRAM, respectively. This is a relatively low occupation level, which enables the use of the largest share of the FPGA for other accelerators.

Program	Description
xdp1	parse pkt headers up to IP, and XDP_DROP
xdp2	parse pkt headers up to IP, and XDP_TX
xdp_adjust_tail	receive pkt, modify pkt into ICMP pkt and XDP_TX
router_ipv4	parse pkt headers up to IP, look up in routing table and forward (redirect)
rxq_info (drop)	increment counter and XDP_DROP
rxq_info (tx)	increment counter and XDP_TX
tx_ip_tunnel	parse pkt up to L4, encapsulate and XDP_TX
redirect_map	output pkt from a specified interface (redirect)

Table 2: Tested Linux XDP example programs.

5 Evaluation

We use a selection of the Linux's XDP example applications and two real world applications to perform the hXDP evaluation. The Linux examples are described in Table 2. The real-world applications are the simple firewall we used as running example, and the Facebook's Katran server load balancer [17]. Katran is a high performance software load balancer that translates virtual addresses to actual server addresses using a weighted scheduling policy, and providing per-flow consistency. Furthermore, Katran collects several flow metrics, and performs IPinIP packet encapsulation.

Using these applications, we perform an evaluation of the impact of the compiler optimizations on the programs' number of instructions, and the achieved level of parallelism. Then, we evaluate the performance of our NetFPGA implementation. In addition, we run a large set of micro-benchmarks to highlight features and limitations of hXDP. We use the microbenchmarks also to compare the hXDP prototype performance with a Netronome NFP4000 SmartNIC. Although the two devices target different deployment scenarios, this can provide further insights on the effect of the hXDP design choices. Unfortunately, the NFP4000 offers only limited eBPF support, which does not allow us to run a complete evaluation. We further include a comparison of hXDP to other FPGA NIC programming solutions, before concluding the section with a brief dicussion of the evaluation results.

5.1 Compiler

Instruction-level optimizations We evaluate the instruction-level optimizations described in Section 3, by activating selectively each of them in the hXDP compiler. Figure 7 shows the reduction of eBPF instructions for a program, relative to its original number of instructions. We can observe that the contribution of each optimization largely depends on the program. For instance, the `xdp_adjust_tail` performs several operations that need to read/write 6B of data, which in turn makes the 6B load/store instructions of our extended ISA particularly effective, providing a 18% reduction in the number of instructions. Likewise, the `simple_firewall` performs several bound checks, which account for 19% of the program's instructions. The parametrized exit reduces the

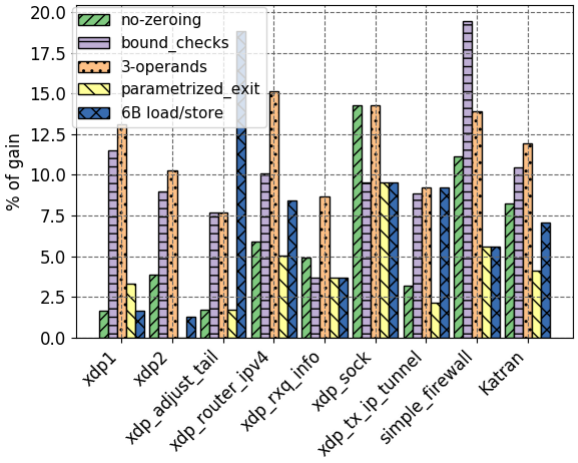


Figure 7: Reduction of instructions due to compiler optimizations, relative to the original number of instructions.

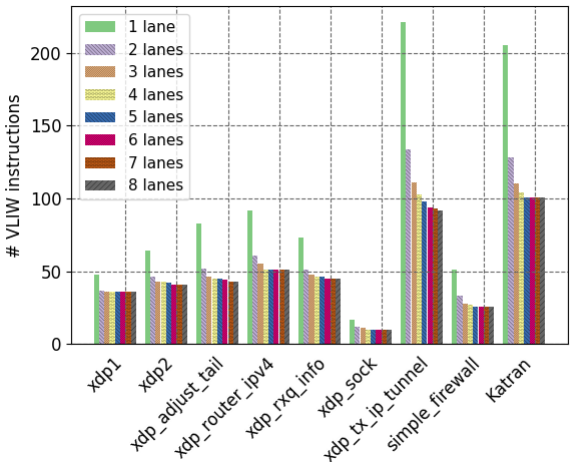


Figure 8: Number of VLIW instructions when varying the available number of execution lanes.

number of instructions by up to 5-10%. However, it should be noted that this reduction has limited impact on the number of instructions executed at runtime, since only one exit instruction is actually executed.

Instructions parallelization We configure the compiler to consider from 2 to 8 parallel execution lanes, and count the number of generated VLIW instructions. A VLIW instruction corresponds to a schedule's row (cf. Section 3.4), and it can therefore contain from 2 to 8 eBPF instructions in this test. Figure 8 shows that the number of VLIW instructions is reduced significantly as we add parallel execution lanes up to 3, in all the cases. Adding a fourth execution lane reduces the VLIW instructions by an additional 5%, and additional lanes provide only marginal benefits. Another important observation is that the compiler's physical register assignment

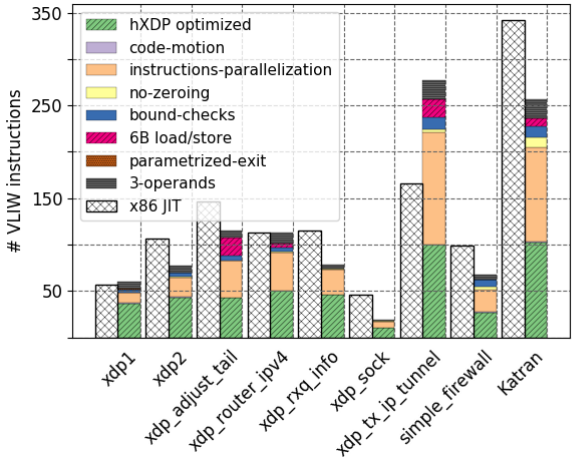


Figure 9: Number of VLIW instructions, and impact of optimizations on its reduction.

step becomes more complex when growing the number of lanes, since there may not be enough registers to hold all the symbols being processed by a larger number of instructions.[9] Given the relatively low gain when growing to more than four parallel lanes, we decided use four parallel lanes in hXDP.

Combined optimizations Figure 9 shows the final number of VLIW instructions produced by the compiler. We show the reduction provided by each optimization as a stacked column, and report also the number of x86 instructions, which result as output of the Linux's eBPF JIT compiler. In Figure, we report the gain for instruction parallelization, and the additional gain from *code movement*, which is the gain obtained by anticipating instructions from control equivalent blocks (cf. Section 3.4). As we can see, when combined, the optimizations do not provide a simple sum of their gains, since each optimization affects also the instructions touched by the other optimizations. Overall, the compiler is capable of providing a number of VLIW instructions that is often 2-3x smaller than the original program's number of instructions. Notice that, by contrast, the output of the JIT compiler for x86 usually grows the number of instructions.[10]

5.2 Hardware performance

We compare hXDP with XDP running on a server machine, and with the XDP offloading implementation provided by a SoC-based Netronome NFP 4000 SmartNIC. The NFP4000 has 60 programmable network processing cores (called *micro-engines*), clocked at 800MHz. The server machine is equipped with an Intel Xeon E5-1630 v3 @3.70GHz, an Intel XL710 40GbE NIC, and running Linux v.5.6.4 with the i40e Intel

[9] This would ultimately require adding more registers, or the introduction of instructions to handle register spilling.

[10] This is also due to the overhead of running on a shared executor, e.g., calling helper functions requires several instructions.

NIC drivers. During the tests we use different CPU frequencies, i.e., 1.2GHz, 2.1GHz and 3.7GHz, to cover a larger spectrum of deployment scenarios. In fact, many deployments favor CPUs with lower frequencies and a higher number of cores [24]. We use a DPDK packet generator to perform throughput and latency measurements. The packet generator is capable of generating a 40Gbps throughput with any packet size and it is connected back-to-back with the system-under-test, i.e., the hXDP prototype running on the NetFPGA, the Netronome SmartNIC or the Linux server running XDP. Delay measurements are performed using hardware packet timestamping at the traffic generator's NIC, and measure the round-trip time. Unless differently stated, all the tests are performed using packets with size 64B belonging to a single network flow. This is a challenging workload for the systems under test. Since we are interested in measuring the hXDP hardware implementation performance, we do not perform tests that require moving packets to the host system. In such cases the packet processing performance would be largely affected by the PCIe and Linux drivers implementations, which are out-of-scope for this paper. We use a similar approach when running tests with the Netronome SmartNIC. As already mentioned, in this case we are only able to run a subset of the evaluation, i.e., some microbenchmarks, due to the the limited eBPF support implemented by the Netronome SmartNICs.

5.2.1 Applications performance

Simple firewall In Section 2 we mentioned that an optimistic upper-bound for the hardware performance would have been 2.8Mpps. When using hXDP with all the compiler and hardware optimizations described in this paper, the same application achieves a throughput of 6.53Mpps, as shown in Figure 10. This is only 12% slower than the same application running on a powerful x86 CPU core clocked at 3.7GHz, and 55% faster than the same CPU core clocked at 2.1GHz. In terms of latency, hXDP provides about 10x lower packet processing latency, for all packet sizes (see Figure 11). This is the case since hXDP avoids crossing the PCIe bus and has no software-related overheads. We omit latency results for the remaining applications, since they are not significantly different.[11] While we are unable to run the simple firewall application using the Netronome's eBPF implementation, Figure 11 shows also the forwarding latency of the Netronome NFP4000 (`nfp` label) when programmed with an XDP program that only performs packet forwarding. Even in this case, we can see that hXDP provides a lower forwarding latency, especially for packets of smaller sizes.

Katran When measuring Katran we find that hXDP is instead 38% slower than the x86 core at 3.7GHz, and only 8% faster than the same core clocked at 2.1GHz. The reason for this relatively worse hXDP performance is the overall program length. Katran's program has many instructions, as such executors with a very high clock frequency are advantaged, since they can run more instructions per second. However, notice the clock frequencies of the CPUs deployed at Facebook's datacenters [24] have frequencies close to 2.1GHz, favoring many-core deployments in place of high-frequency ones. hXDP clocked at 156MHz is still capable of outperforming a CPU core clocked at that frequency.

Linux examples We finally measure the performance of the Linux's XDP examples listed in Table 2. These applications allow us to better understand the hXDP performance with programs of different types (see Figure 12). We can identify three categories of programs. First, programs that forward packets to the NIC interfaces are faster when running on hXDP. These programs do not pass packets to the host system, and thus they can live entirely in the NIC. For such programs, hXDP usually performs at least as good as a single x86 core clocked at 2.1GHz. In fact, processing XDP on the host system incurs the additional PCIe transfer overhead to send the packet back to the NIC. Second, programs that always drop packets are usually faster on x86, unless the processor has a low frequency, such as 1.2GHz. Here, it should be noted that such programs are rather uncommon, e.g., programs used to gather network traffic statistics receiving packets from a network tap. Finally, programs that are long, e.g., `tx_ip_tunnel` has 283 instructions, are faster on x86. Like we noticed in the case of Katran, with longer programs the hXDP's implementation low clock frequency can become problematic.

5.2.2 Microbenchmarks

Baseline We measure the baseline packet processing performance using three simple programs: *XDP_DROP* drops the packet as soon as it is received; *XDP_TX* parses the Ethernet header and swaps MAC addresses before sending the packet out to the port from which it was received; *redirect* is like XDP_TX, but sends the packet out to a different port, which requires calling a specific XDP helper function. The performance results clearly show the advantage of running on the NIC and avoiding PCIe transfers when processing small programs (see Figure 13). hXDP can drop 52Mpps vs the 38Mpps of the x86 CPU core@3.7GHz, and 32Mpps of the Netronome NFP4000. Here, the very high performance of hXDP is due to the parametrized exit/early exit optimizations mentioned in Section 4. Disabling the optimization brings down the hXDP performance to 22Mpps. In the case of XDP_TX, instead, hXDP forwards 22.5Mpps while x86 can forward 12Mpps. The NFP4000 is the fastest device in this test, forwarding over 28Mpps. In the case of redirect, hXDP provides 15Mpps, while x86 can only forward 11Mpps when running at 3.7GHz. Here, the redirect has lower performance because eBPF im-

[11] The impact of different programs is especially significant with small packet sizes. However, even in such cases we cannot observe significant differences. In fact each VLIW instruction takes about 7 nanoseconds to be executed, thus, differences of tens of instructions among programs change the processing latency by far less than a microsecond.

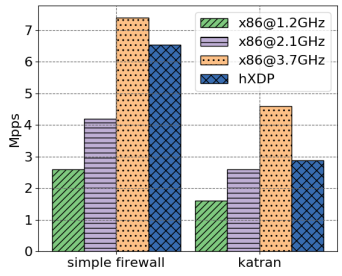


Figure 10: Throughput for real-world applications. hXDP is faster than a high-end CPU core clocked at over 2GHz.

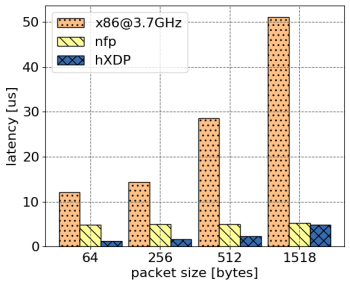


Figure 11: Packet forwarding latency for different packet sizes.

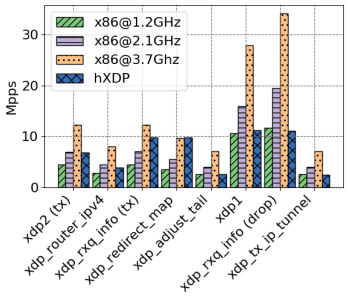


Figure 12: Throughput of Linux's XDP programs. hXDP is faster for programs that perform TX or redirection.

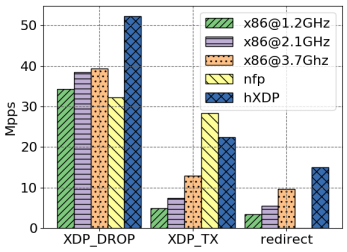


Figure 13: Baseline throughput measurements for basic XDP programs.

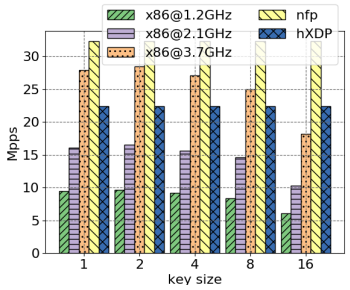


Figure 14: Impact on forwarding throughput of map accesses.

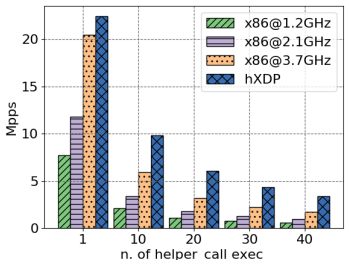


Figure 15: Forwarding tput when calling a helper function.

plements it with a helper. Results for the NFP4000 are not available since the it does not support the redirect action.

Maps access Accessing eBPF maps affects the performance of XDP programs that need to read and keep state. In this test we measure the performance variation when accessing a map with a variable key size ranging between 1-16B. Accessing the map is performed calling a helper function that performs a hash of the key and then retrieves the value from memory. In the x86 tests we ensure that the accessed entry is in the CPU cache. Figure 14 shows that the hXDP prototype has constant access performance, independently from the key size. This is the result of the wider memory data buses, which can in fact accomodate a memory access in a single clock cycle for keys of up to 32B in size. The NFP4000, like hXDP, shows no performance drop when increasing the key size. Instead, in the x86 case the performance drops when the key size grows from 8B to 16B, we believe this is due to the need to perform multiple loads.

Helper functions In this micro-benchmark we measure throughput performance when calling from 1 to 40 times a helper function that performs an incremental checksum calculation. Since helper functions are implemented as dedicated hardware functions in hXDP, we expect our prototype to exhibit better performance than x86, which is confirmed by our results (see Figure 15). I.e., hXDP may provide significant benefits when offloading programs that need complex computations captured in helper functions. Also, the hXDP's helper function machinery may be used to eventually replace sets of common instructions with more efficient dedicated hardware implementations, providing an easier pathway for future extensions of the hardware implementation.

Program	# instr.	x86 IPC	hXDP IPC
xdp1	61	2.20	1.70
xdp2	78	2.19	1.81
xdp_adjust_tail	117	2.37	2.72
router_ipv4	119	2.38	2.38
rxq_info	81	2.81	1.76
tx_ip_tunnel	283	2.24	2.83
simple_firewall	72	2.16	2.66
Katran	268	2.32	2.62

Table 3: Programs' number of instructions, x86 runtime instruction-per-cycle (IPC) and hXDP static IPC mean rates.

Instructions per cycle We compare the parallelization level obtained at compile time by hXDP, with the runtime parallelization performed by the x86 CPU core. Table 3 shows that the static hXDP parallelization achieves often a parallelization level as good as the one achieved by the complex x86 runtime machinery.[12]

5.2.3 Comparison to other FPGA solutions

hXDP provides a more flexible programming model than

[12]The x86 IPC should be understood as a coarse-grained estimation of the XDP instruction-level parallelism since, despite being isolated, the CPU runs also the operating system services related to the eBPF virtual machine, and its IPC is also affected by memory access latencies, which more significantly impact the IPC for high clock frequencies.

previous work for FPGA NIC programming. However, in some cases, simpler network functions implemented with hXDP could be also implemented using other programming approaches for FPGA NICs, while keeping functional equivalence. One such example is the simple firewall presented in this paper, which is supported also by FlowBlaze [45].

Throughput Leaving aside the cost of re-implementing the function using the FlowBlaze abstraction, we can generally expect hXDP to be slower than FlowBlaze at processing packets. In fact, in the simple firewall case, FlowBlaze can forward about 60Mpps, vs the 6.5Mpps of hXDP. The FlowBlaze design is clocked at 156MHz, like hXDP, and its better performance is due to the high-level of specialization. FlowBlaze is optimized to process only packet headers, using statically-defined functions. This requires loading a new bitstream on the FPGA when the function changes, but it enables the system to achieve the reported high performance.[13] Conversely, hXDP has to pay a significant cost to provide full XDP compatibility, including dynamic network function programmability and processing of both packet headers and payloads.

Hardware resources A second important difference is the amount of hardware resources required by the two approaches. hXDP needs about 18% of the NetFPGA logic resources, independently from the particular network function being implemented. Conversely, FlowBlaze implements a packet processing pipeline, with each pipeline's stage requiring about 16% of the NetFPGA's logic resources. For example, the simple firewall function implementation requires two FlowBlaze pipeline's stages. More complex functions, such as a load balancer, may require 4 or 5 stages, depending on the implemented load balancing logic [19].

In summary, the FlowBlaze's pipeline leverages hardware parallelism to achieve high performance. However, it has the disadvantage of often requiring more hardware resources than a sequential executor, like the one implemented by hXDP. Because of that, hXDP is especially helpful in scenarios where a small amount of FPGA resources is available, e.g., when sharing the FPGA among different accelerators.

5.3 Discussion

Suitable applications hXDP can run XDP programs with no modifications, however, the results presented in this section show that hXDP is especially suitable for programs that can process packets entirely on the NIC, and which are no more than a few 10s of VLIW instructions long. This is a common observation made also for other offloading solutions [26].

FPGA Sharing At the same time, hXDP succeeds in using little FPGA resources, leaving space for other accelerators. For instance, we could co-locate on the same FPGA several instances of the VLDA accelerator design for neural networks presented in [12]. Here, one important note is about the use of memory resources (BRAM). Some XDP programs may need larger map memories. It should be clear that the memory area dedicated to maps reduces the memory resources available to other accelerators on the FPGA. As such, the memory requirements of XDP programs, which are anyway known at compile time, is another important factor to consider when taking program offloading decisions.

[13] FlowBlaze allows the programmer to perform some runtime reconfiguration of the functions, however this a limited feature. For instance, packet parsers are statically defined.

6 Future work

While the hXDP performance results are already good to run real-world applications, e.g., Katran, we identified a number of optimization options, as well as avenues for future research.

Compiler First, our compiler can be improved. For instance, we were able to hand-optimize the simple firewall instructions and run it at 7.1Mpps on hXDP. This is almost a 10% improvement over the result presented in Section 5. The applied optimizations had to do with a better organization of the memory accesses, and we believe they could be automated by a smarter compiler.

Hardware parser Second, XDP programs often have large sections dedicated to packet parsing. Identifying them and providing a dedicated programmable parser in hardware [23] may significantly reduce the number of instructions executed by hXDP. However, it is still unclear what would be the best strategy to implement the parser on FPGA and integrate it with hXDP, and the related performance and hardware resources usage trade offs.

Multi-core and memory Third, while in this paper we focused on a single processing core, hXDP can be extended to support two or more `Sephirot` cores. This would effectively trade off more FPGA resources for higher forwarding performance. For instance, we were able to test an implementation with two cores, and two lanes each, with little effort. This was the case since the two cores shared a common memory area and therefore there were no significant data consistency issues to handle. Extending to more cores (lanes) would instead require the design of a more complex memory access system. Related to this, another interesting extension to our current design would be the support for larger DRAM or HBM memories, to store large memory maps.

ASIC Finally, hXDP targets FPGA platforms, since we assume that FPGAs are already available in current deployments. Nonetheless, hXDP has several fixed design's components, such as the `Sephirot` core, which suggests that hXDP may be implemented as ASIC. An ASIC could provide a potentially higher clock frequency, and an overall more efficient use of the available silicon resources. Here, in addition to measuring the performance that such a design may achieve, there are additional interesting open questions. For instance evaluating the potential advantages/disadvantages provided by the ability

to change helper functions implemented in the FPGA, when compared to a fixed set of helper functions provided in ASIC.

7 Related Work

NIC Programming AccelNet [18] is a match-action offloading engine used in large cloud datacenters to offload virtual switching and firewalling functions, implemented on top of the Catapult FPGA NIC [10]. FlexNIC [32] is a design based on the RMT [8] architecture, which provides a flexible network DMA interface used by operating systems and applications to offload stateless packet parsing and classification. P4->NetFPGA [1] and P4FPGA [56] provide high-level synthesis from the P4 [7] domain-specific language to an FPGA NIC platform. FlowBlaze [45] implements a finite-state machine abstraction using match-action tables on an FPGA NIC, to implement simple but high-performance network functions. Emu [50] uses high level synthesis to implement functions described in C# on the NetFPGA. Compared to these works, instead of match-action or higher-level abstractions, hXDP leverages abstractions defined by the Linux's kernel, and implements network functions described using the eBPF ISA.

The Netronome SmartNICs implement a limited form of eBPF/XDP offloading [33]. Unlike hXDP that implements a solution specifically targeted to XDP programs, the Netronome solution is added on top of their network processor as an afterthought, and therefore it is not specialized for the execution of XDP programs.

Application frameworks AccelTCP [38], Tonic [2] and Xtra [5] present abstractions, hardware architectures and prototypes to offload the transport protocol tasks to the NIC. We have not investigated the feasibility of using hXDP for a similar task, which is part of our future work. NICA [16] and ClickNP [36] are software/hardware frameworks that introduce specific software abstractions that connect FPGA blocks with an user program running on a general purpose CPU. In both cases, applications can only be designed composing the provided hardware blocks. hXDP provides instead an ISA that can be flexibly programmed, e.g., using higher level languages such as C.

Applications Examples of applications implemented on NICs include: DNS resolver [57]; the paxos protocol [51]; network slicing [59]; Key-value stores [34, 35, 49, 52]; Machine Learning [14, 21, 41]; and generic cloud services as proposed in [10, 46, 48]. [37] uses SmartNICs to provide a microservice-based platform to run different services. In this case, SoC-based NICs are used, e.g., based on Arm processing cores. Lynx [53] provides a system to implement network-facing services that need access to accelerators, such as GPUs, without involving the host system's CPU. Floem [43] is a framework to simplify the design of applications that leverage offloading to SoC-based SmartNICs. hXDP provides an XDP programming model that can be used to implement and extend these applications.

NIC Hardware Previous work presenting VLIW core designs for FPGAs did not focus on network processing [29, 31]. [9] is the closest to hXDP. It employs a non-specialized MIPS-based ISA and a VLIW architecture for packet processing. hXDP has an ISA design specifically targeted to network processing using the XDP abstractions. [20] presents an open-source 100-Gbps FPGA NIC design. hXDP can be integrated in such design to implement an open source FPGA NIC with XDP offloading support.

8 Conclusion

This paper presented the design and implementation of hXDP, a system to run Linux's XDP programs on FPGA NICs. hXDP can run unmodified XDP programs on FPGA matching the performance of a high-end x86 CPU core clocked at more than 2GHz. Designing and implementing hXDP required a significant research and engineering effort, which involved the design of a processor and its compiler, and while we believe that the performance results for a design running at 156MHz are already remarkable, we also identified several areas for future improvements. In fact, we consider hXDP a starting point and a tool to design future interfaces between operating systems/applications and network interface cards/accelerators. To foster work in this direction, we make our implementations available to the research community.[14]

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Costin Raiciu for their extensive and valuable feedback and comments, which have substantially improved the content and presentation of this paper.

The research leading to these results has received funding from the ECSEL Joint Undertaking in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National Authorities, under grant agreement n. 876967 (Project "BRAINE").

References

[1] P4-NetFPGA. https://github.com/NetFPGA/P4-NetFPGA-public/wiki.

[2] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, Santa Clara, CA, Feb. 2020. USENIX Association.

[14] https://github.com/axbryd

[3] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[4] G. Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, 2017.

[5] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, and S. Pontarelli. XTRA: Towards portable transport layer functions. *IEEE Transactions on Network and Service Management*, 16(4):1507–1521, 2019.

[6] S. Biookaghazadeh, M. Zhao, and F. Ren. Are fpgas suitable for edge computing? In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, Boston, MA, July 2018. USENIX Association.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[9] M. S. Brunella, S. Pontarelli, M. Bonola, and G. Bianchi. V-PMP: A VLIW packet manipulator processor. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–9. IEEE, 2018.

[10] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[11] V. Chamola, S. Patra, N. Kumar, and M. Guizani. Fpga for 5g: Re-configurable hardware for next generation communication. *IEEE Wireless Communications*, pages 1–8, 2020.

[12] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, Oct. 2018. USENIX Association.

[13] D. Chiou. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 124–124. IEEE, 2017.

[14] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[15] M. V. Dumitru, D. Dumitrescu, and C. Raiciu. Can we exploit buggy p4 programs? In *Proceedings of the Symposium on SDN Research*, SOSR '20, page 62–68, New York, NY, USA, 2020. Association for Computing Machinery.

[16] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 345–362, Renton, WA, July 2019. USENIX Association.

[17] Facebook. Facebook. 2018. Katran source code repository. https://github.com/facebookincubator/katran.

[18] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, Apr. 2018. USENIX Association.

[19] FlowBlaze. Repository with FlowBlaze source code and additional material. http://axbryd.com/FlowBlaze.html.

[20] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020.

[21] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams,

M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[22] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[23] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, page 13–24. IEEE Press, 2013.

[24] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at Facebook: a datacenter infrastructure perspective. In *High Performance Computer Architecture (HPCA)*. IEEE, 2018.

[25] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch. A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2019.

[26] O. Hohlfeld, J. Krude, J. H. Reelfs, J. Rüth, and K. Wehrle. Demystifying the performance of XDP BPF. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 208–212. IEEE, 2019.

[27] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Intel. 5g wireless. https://www.intel.com/content/www/us/en/communications/products/programmable/applications/baseband.html, 2020.

[29] C. Iseli and E. Sanchez. Spyder: A reconfigurable vliw processor using FPGAs. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 17–24. IEEE, 1993.

[30] S. Jiang, D. He, C. Yang, C. Xu, G. Luo, Y. Chen, Y. Liu, and J. Jiang. Accelerating mobile applications at the network edge with software-programmable fpgas. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 55–62. IEEE, 2018.

[31] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster. An fpga-based vliw processor with custom hardware execution. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, FPGA '05, page 107–117, New York, NY, USA, 2005. Association for Computing Machinery.

[32] A. Kaufmann, S. Peter, T. Anderson, and A. Krishnamurthy. Flexnic: Rethinking network DMA. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[33] J. Kicinski and N. Viljoen. eBPF hardware offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev*, 1, 2016.

[34] M. Lavasani, H. Angepat, and D. Chiou. An FPGA-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 13(2):57–60, 2013.

[35] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137–152, New York, NY, USA, 2017. Association for Computing Machinery.

[36] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[37] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, Renton, WA, July 2019. USENIX Association.

[38] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. Acceltcp: Accelerating network applications with stateful TCP offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, Santa Clara, CA, Feb. 2020. USENIX Association.

[39] NEC. Building an Open vRAN Ecosystem White Paper. https://www.nec.com/en/global/solutions/5g/index.html, 2020.

[40] Netronome. AgilioTM CX 2x40GbE intelligent server adapter. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf.

[41] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss, and E. S. Chung. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–38, 2015.

[42] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–38, 2015.

[43] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, Oct. 2018. USENIX Association.

[44] S. Pinneterre, S. Chiotakis, M. Paolino, and D. Raho. vfpgamanager: A virtualization framework for orchestrated fpga accelerator sharing in 5g cloud environments. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5. IEEE, 2018.

[45] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, Feb. 2019. USENIX Association.

[46] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 13–24. IEEE Press, 2014.

[47] R. Ricart-Sanchez, P. Malagon, P. Salva-Garcia, E. C. Perez, Q. Wang, and J. M. A. Calero. Towards an fpga-accelerated programmable data path for edge-to-core communications in 5g networks. *Journal of Network and Computer Applications*, 124:80–93, 2018.

[48] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 150–156, New York, NY, USA, 2017. Association for Computing Machinery.

[49] G. Siracusano and R. Bifulco. Is it a smartnic or a key-value store? both! In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, page 138–140, New York, NY, USA, 2017. Association for Computing Machinery.

[50] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 459–471, Santa Clara, CA, July 2017. USENIX Association.

[51] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Y. Tokusashi, H. Matsutani, and N. Zilberman. Lake: the power of in-network computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2018.

[53] M. Tork, L. Maudlej, and M. Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 117–131, New York, NY, USA, 2020. Association for Computing Machinery.

[54] W. Tu, J. Stringer, Y. Sun, and Y.-H. Wei. Bringing the power of ebpf to open vswitch. In *Linux Plumbers Conference*, 2018.

[55] F. J. Villanueva, M. J. Santofimia, D. Villa, J. Barba, and J. C. Lopez. Civitas: The smart city middleware, from sensors to big data. In *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 445–450. IEEE, 2013.

[56] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 122–135, New York, NY, USA, 2017. Association for Computing Machinery.

[57] J. Woodruff, M. Ramanujam, and N. Zilberman. P4DNS: In-network DNS. In *2019 ACM/IEEE Symposium on*

Architectures for Networking and Communications Systems (ANCS), pages 1–6. IEEE, 2019.

[58] Xilinx. 5G Wireless Solutions Powered by Xilinx. https://www.xilinx.com/applications/megatrends/5g.html, 2020.

[59] Y. Yan, A. F. Beldachi, R. Nejabati, and D. Simeonidou. P4-enabled smart nic: Enabling sliceable and service-driven optical data centres. *Journal of Lightwave Technology*, 38(9):2688–2694, 2020.

[60] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro '14*, 34(5):32–41, 2014.

Appendix: Artifacts

Source code, examples and instructions to replicate the results presented in this paper are provided at https://github.com/axbryd/hXDP-Artifacts.



Do OS abstractions make sense on FPGAs?

Dario Korolija, Timothy Roscoe, Gustavo Alonso
Systems Group, Dept. of Computer Science, ETH Zurich
{dario.korolija, troscoe, alonso}@inf.ethz.ch

Abstract

Hybrid computing systems, consisting of a CPU server coupled with a Field-Programmable Gate Array (FPGA) for application acceleration, are today a common facility in datacenters and clouds. FPGAs can deliver tremendous improvements in performance and energy efficiency for a range or workloads, but development and deployment of FPGA-based applications remains cumbersome, leading to recent work which replicates subsets of the traditional OS execution environment (virtual memory, processes, etc.) on the FPGA.

In this paper we ask a different question: to what extent do traditional OS abstractions make sense in the context of an FPGA as part of a hybrid system, particularly when taken as a *complete package,* as they would be in an OS? To answer this, we built and evaluated Coyote, an open source, portable, configurable "shell" for FPGAs which provides a full suite of OS abstractions, working with the host OS. Coyote supports secure spatial and temporal multiplexing of the FPGA between tenants, virtual memory, communication, and memory management inside a uniform execution environment. The overhead of Coyote is small and the performance benefit is significant, but more importantly it allows us to reflect on whether importing OS abstractions wholesale to FPGAs is the best way forward.

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are now standard in datacenters and cloud providers [1, 3, 12], providing more flexibility at lower power than ASICs or GPUs for many applications (e.g. [5, 19, 25, 29, 30, 41, 53]) despite (due to their heritage in embedded systems and prototyping) remaining difficult to program, deploy, and securely manage. As a result, along with much research into making FPGAs easier to program [7, 8, 36, 45, 51, 54, 58], considerable recent work applied ideas from operating systems design and implementation to resource allocation, sharing, isolation, and management of an FPGA-centric computer.

So far, this work has been piecemeal, focusing on a particular aspect of functionality, e.g. Feniks [63] targets FPGA access to peripherals, Optimus [32] provides access to a host's virtual memory via address translation, etc. These yield substantial incremental improvements over the state of the art.

At the same time, what makes good OS design so challenging is the close interaction in the kernel between *all* the functionality. Virtual memory without support for multiple applications (multi-tenancy) or strong isolation between them is of limited use. Virtualizing hardware devices without providing virtual addressing and creating a common execution enviroment that abstracts the hardware leaves most of the problem unsolved. An FPGA scheduler that cannot exploit the ability to dynamically reconfigure parts of the chip has a limited shelf-life, and so on.

Therefore, we step back to ask the question: to what extent can (or should) traditional OS concepts (processes, virtual memory, etc.) be usefully translated to an FPGA? What happens when they are? To answer this question, we need to adopt a comprehensive, holistic approach and think about complete functionality, rather than sticking to particular aspects of an OS or supporting only limited FPGA features.

To this end, we have built Coyote, combining a coherent set of OS abstractions in a single unified runtime for FPGA-based applications. Like a microkernel, Coyote provides the core set of essential functions on which other services can be based: a uniform execution environment and portability layer, virtual memory, physical memory management, communication, spatial and temporal scheduling, networking, and an analog of software processes or tasks for user logic. It achieves this with minimal overhead (less than 15% of a commodity FPGA). Our contributions in this paper are therefore:

1. For a range of OS abstractions, a critical assessment of how each might map to an FPGA, in the context of its interaction with the others,

2. An implementation of the complete ensemble in Coyote, a configurable FPGA "OS" for hybrid compute servers.

3. A quantitative evaluation of Coyote using both microbenchmarks and 5 real applications.

4. A qualitative discussion of the implications of the work for future FPGA and OS designs.

We start with the basic hardware that any FPGA OS must handle. This determines the high-level structure of Coyote.

2 Foundations

Coyote targets hybrid systems, combining a conventional CPU with an FPGA either over a peripheral bus like PCIe, CXL [16], CCIX [13] or OpenCAPI [49], or instead a native coherency protocol as with Intel HARP [39] or ETH Enzian [2, 21]. Coyote runs today on PCs with Xilinx VCU118 [56], Alveo U250 [59] and Alveo U280 [60] cards. The port to Enzian is under way. We avoid any design decisions that might prevent the use of modern FPGA features like dynamic partial reconfiguration of multiple regions, or useful "hard" on-chip functions.

This naturally splits any design into a "hardware" component running on the FPGA and a software component running on the host CPU as part of the OS and support libraries.

Furthermore, dynamic reconfiguration of the FPGA induces a further split of the hardware component into a "static region", configured at boot, and a "dynamic region", containing subregions (vFPGAs), each of which may be changed on the fly. This split exists (often in simplified form) in all FPGA datacenter deployments. Within and between regions, hardware components interact via standard interconnects like AXI [31].

2.1 The static region

The FPGA static region must contain the functionality required to reconfigure the dynamic region and communicate with the CPU's OS. However, its contents should not be fixed for all time. Space (chip area, logic blocks, wires, etc.) remains a scarce resource on FPGAs, and unlike OS resources such as CPU time and virtual memory, it is hard to make it "elastic" through virtualization. Moreover, different models of FPGAs show very different tradeoffs. In the medium term, it is important to make some static region components (for example, the TCP/IP stack) optional so they can be omitted if the space is better used for user logic.

In Coyote, the static region always contains logic to partially reconfigure the dynamic region, communicate with the host machine (an xDMA copy engine [57]), and to divide the dynamic region into a set of *virtual FPGAs* ("vFPGAs"), each of which has an interface mapped into the physical address space of the host CPU (described below).

The static region can also contain optional logic shared between all applications running in vFPGAs, the most basic being memory controllers (for RAM directly connected to the FPGA) and networking (at present, TCP and RDMA).

2.2 The dynamic region

The dynamic region is the basic mechanism for time-division multiplexing of the FPGA resources. Modern FPGAs allow selective portions of this region to be reconfigured independently at any time. Most deployed systems (e.g. F1 [3] and Intel's HARP [39]) dedicate this region to a single application, and reprogram it only rarely (e.g. when an associated virtual machine on the host is booted up).

Coyote, like other recent systems [14, 17, 62, 63], provides flexible spatial and temporal multiplexing. The dynamic region is partitioned into independent vFPGAs. Their number is wired into the static region, which allows multiple applications to run concurrently and be switched in and out.

A novel feature of Coyote is that each vFPGA is further divided into *user logic* and a *wrapper*. The former is a bitstream entirely synthesized by a Coyote user and validated by the system. This allows great flexibility in programming models: Coyote applications can be written in HLS, Verilog, VHDL, OpenCL, or some combination of these or other languages.

The wrapper is part of Coyote, and both sandboxes user logic and provides a standard interface to the rest of the system (in FPGA terms, partition pins are inserted by the reconfiguration tool locking all the boundary interface signals in the fabric). This incurs a cost in chip area usage, but the benefit is that Coyote pushes the "portability layer" for FPGA applications up to the language level: an application written for Coyote can, given sufficient resources, be synthesized to run on any Coyote FPGA. In contrast, with native FPGA development at present code is rarely portable between device models (or even, in some cases, revisions of the same model).

It is tempting to draw an analogy between the structure of Coyote and a microkernel model of an OS, consisting of the kernel (the static region), services (optional static components), system libraries (dynamic wrappers), and applications code (user logic). However, this would be an error. For example, the dynamic wrappers form part of a trusted computing base (TCB), whereas system libraries in a microkernel do not.

2.3 The software component

In a hybrid system, the host OS must clearly be aware of the FPGA environment, and also provide suitable and safe abstractions to user application code running on the CPU for interacting with user logic on the FPGA.

Beyond this, however, there is a fundamental tradeoff between how much management of FPGA resources is performed on the FPGA itself (by a combination of static region logic and dynamic functionality) and how much is implemented by system software on the CPU. Offloading FPGA

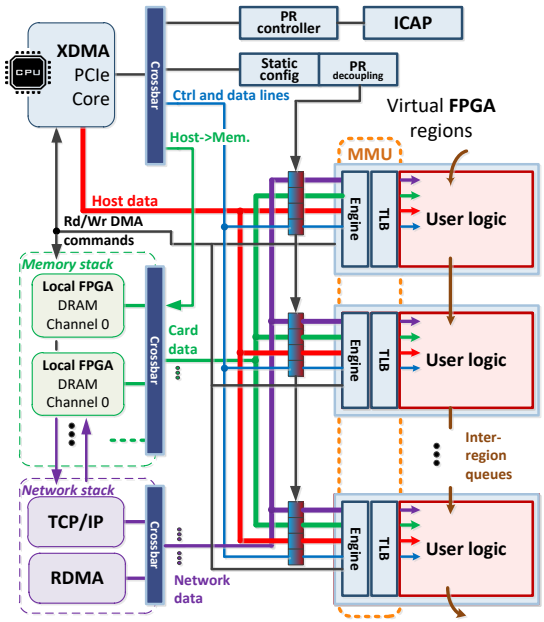


Figure 1: Coyote structure

management functionality to the CPU and OS frees up valuable space on the FPGA, and allows much more policy flexibility than could be reasonably implemented in logic. Hovewer a functionality that is on a critical path can lead to degraded performance and/or loss of predictability in response time (often a key attribute of hardware solutions). In some ways this mirrors the traditional OS tradeoff between kernel-mode and user-space implementation, but the contrast is more stark.

Coyote maximises the FPGA area available to user logic, moving much functionality not on the fast path into the host CPU's OS. The software part of Coyote consists of a kernel *driver* (currently for Linux), a *runtime manager* process, and user application code.

At startup, the driver reads the configuration of the static region from the FPGA and sets up data structures for the set of vFPGAs to be supported. Thereafter, it is responsible for "control plane" communication with the FPGA (such as reconfiguring a vFPGA) and creating memory mappings for application code to interact directly with a vFPGA. The driver also handles dynamic memory allocation for the FPGA, and services TLB misses on the FPGA (see below).

Figure 1 shows the components of Coyote. The host CPU is connected to the PCIe core at top left.

3 OS abstractions on an FPGA

In this section, for each considered OS abstraction we first review its role in a conventional OS for a homogeneous multicore machine. We then discuss what is fundamentally different in an FPGA environment, and the impact this has on design decisions when creating an analog of the abstraction on the FPGA. Following this, we discuss our own implementation, and discuss our experience with building and using the approach. A quantitative evaluation of the whole of Coyote is given in Section 4.

3.1 Processes, tasks, and threads

The basic abstractions most OSes provide for multiplexing and virtualizing processor resources are based on processes, threads, and/or tasks. Definitions vary from OS to OS, but a thread is generally an open-ended execution of a sequence of instructions on a single virtual processor, a task is a unit of computational work to be dispatched to a CPU core, and a process is some combination of threads sharing an address space, to which CPU resources are allocated.

The hardware mechanisms underlying these abstractions are basically the ability of the processor to context switch, and be preempted by an interrupt or trap.

Such abstractions can be readily adapted to architectures like GPUs, which retain the notion of a hardware thread, albeit with a very different degree of parallelism. GPU drivers for modern OSes attempt to extend the process abstraction of the host CPU to the GPU, although in a somewhat limited form [9], and this is the foundation for programming models like CUDA and OpenCL. The task abstraction has also been successfully deployed on GPUs [44].

What's different on an FPGA? Resource multiplexing on FPGAs is fundamentally different, since there is no hardware corresponding to a "processor", "core", or "hardware thread" on which to base an abstraction aimed at multiplexing processing resources. Instead, the basic mechanisms available on the FPGA for multiplexing compute resources between principals are partial reconfiguration of areas of the FPGA logic at runtime, and spatial partitioning of application logic across different areas of the chip.

While it is true that a popular programming technique for FPGAs involves implementing a custom application-specific processor (typically some VLIW-based architecture), this is not intended to be multiplexed or scheduled. The analogue of these custom cores in the software world is more that of a library or bytecode interpreter that lives entirely within the process abstraction.

The trivial approach here is to dedicate the entire FPGA to a single application, and indeed in embedded systems this is the norm. A more flexible approach allows more than one application to use the FPGA at a time. The static region of the FPGA contains enough logic to swap one application out for another, but otherwise the chip is dedicated to an application for long periods. This is the model adopted by Amazon F1 and, indeed, almost all other commerically deployed systems.

An alternative proposed in research systems (e.g. [32, 62, 63] and others) is to partition the FPGA resources statically

between applications. Spatial partitioning also raises further questions. For example, when multiple applications share the FPGA, should they be allowed to communicate, as processes do with IPC, and if so, how?

Coyote approach: Coyote combines both approaches, providing a cooperative multitasking abstraction of a set of virtual FPGAs, each of which is timeshared between applications. A Coyote platform is configured at boot time with a fixed number (e.g. 2-8) of vFPGAs, which are a spatial partition of the dynamic region of the chip. Each of these regions are, for the purposes of executing user logic, equivalent (much like cores in a symmetric multiprocessor), and are time-shared between applications. Ideally, a single application bitstream could be loaded into any available "slot" to be executed. Although some research in this direction exists [20], this is difficult with current levels of heterogeneity in FPGAs, which means that (at present) each application has to be (automatically) synthesized in advance for each vFPGA slot, akin to compiling "fat binaries" for multiple architectures. We discuss specific spatial and temporal scheduling questions below, along with the execution environment provided to user logic.

Discussion: A scheme with this generality requires care to implement. When timesharing vFPGAs, it is important that the context switch overhead does not outweigh the performance benefits of using a circuit in the first place. Dynamic partial reconfiguration of an FPGA is a relatively slow process and may remain so for the foreseeable future. In 4.3 we measure this cost.

Moreover, the logic required to implement multiple vFPGAs, and allow them to communicate and share services in the static region of the chip, must come with an acceptably-small cost in chip resources. We evaluate this in Section 4.2. So far our experience has been good: we can comfortably run multiple useful applications on a single FPGA today, and hardware trends are in our favour as the parts become larger.

3.2 Execution environment

The process abstraction also serves the purpose of providing a standard *execution environment* for a program. A program compiled to run in a process can, in principle, execute in any process on any machine implementing the same process environment. For example, in Unix, a process's execution environment consists of a virtual address space, one or more threads, a set of file descriptors, the system call interface, etc.

What's different on an FPGA? To date, there are almost no attempts to define a process-like execution environment for an FPGA. Most FPGA application development targets a specific model of FPGA. Porting the same logic to a different chip is often a non-trivial programming problem.

The heterogeneous nature of hybrid platforms complicates this question further. In addition to the environment in which user logic executes, a process abstraction must also address how software processes and FPGA-based logic "processes" interact across the hardware/software interface.

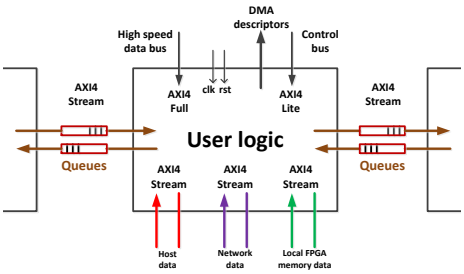


Figure 2: User logic interface

In GPUs, programming models like OpenCL and CUDA are the solution. Portability is raised to the compiler, and the execution environment is defined by the language in which the GPU code is written. This works well for GPUs because they function as pure accelerators. The same model has been implemented for FPGAs [51, 58].

However, hybrid FPGA-based systems are not pure computational devices - for example, they perform I/O through network and storage interfaces; indeed, this ability to interface externally is a major selling point. Rolling this functionality entirely into a compiler has not worked in conventional machines, and is unlikely to do so here. Instead, runtime interfaces are needed. Perhaps the closest GPU analogy here is ptasks [44], which present the GPU as a task-based runtime as opposed to a language-level OpenCL interpreter.

Coyote approach: Coyote defines a single *user logic interface* (ULI) for every application, which is the hardware analog of an ABI, and is illustrated in Figure 2. It uses the streaming AXI4 protocol for transferring bulk data between the host, memory stack, other services like the network stack, and the user logic. The same interface is used for inter-region communication, with a control plane over an AXI4-light bus.

This interface is provided by the dynamic wrapper in each vFPGA, and effectively sandboxes the user logic while providing communication with system services and memory – effectively combining functions of an address space and system call ABI in a software process.

Access to the ULI interface is exposed to user logic at a fairly low level, allowing read and write descriptors to be generated directly from the user logic in the FPGA fabric, and host software access (including by high-bandwidth SIMD instructions) to be routed directly to the user logic.

User software on the CPU interacts with the FPGA by creating a *job object*, essentially a closure consisting of user logic and other parameters and data. This is passed to the runtime manager for installation on the FPGA.

Once functional, the user logic exposes a register interface in physical memory to the CPU, and the runtime manager maps this into the calling process' address space. Thereafter, the interraction between application software and user logic

completely bypasses the kernel and runtime manager.

Discussion: The ULI in Coyote incurs minimal overhead, but delivers considerable benefits, some of which might be surprising to those familiar with software development. It enables an approach analogous to microkernels, with common services provided to multiple vFPGAs over the AXI interconnect. For example, we ported publicly-available TCP and RoCE stacks [46, 48] to Coyote, and they became immediately usable to our existing applications without the extensive hardware-specific modifications usually required in FPGA development.

As with conventional OSes, the Coyote execution environment also provides a way to deal with the evolution of hardware. The FPGA design space is changing rapidly. To take one example: in most FPGAs deployed in data centers today, the memory controllers and network stack (aside from PHY and MAC) are still instantiated as reconfigurable logic. However, both are becoming an almost universal requirement for cloud FPGA applications, which makes a strong case for building "hard" IP into future FPGAs to provide this functionality with less penalty in chip area - indeed, the latest design of Microsoft's Catapult platform offloads the network stack to an ASIC (albeit off-chip). Intel's Embedded Multi-Die Interconnect Bridge (EMIB) is intended to extend FPGAs with new hardware, for example machine learning accelerators [37]. Recent Xilinx Versal cards also provide numerous off-chip hardware functions.

These trends make it even more difficult to achieve portability without a uniform execution environment like Coyote's to abstract these features behind a clean interface.

3.3 Scheduling

Scheduling on conventional machines is a complex topic with a history older than computers themselves. In this paper we focus on factors affecting scheduling mechanisms rather than specific policies.

What's different on an FPGA? CPU scheduling can be preemptive or non-preemtive. Preemptive scheduling on CPUs requires a mechanism to interrupt a running process, save its state, and context switch to another, without any cooperation from the process or user program itself.

On an FPGA, such interrupt machanisms are not supported by any of the mainstream toolchains. Some progress in this direction has been made in academia [27], but with significant performance penalties and implementation difficulties. Furthermore, the "state" of executing user logic potentially includes any stateful logic block (block RAM, flip-flops, DSPs) in the region of the FPGA used by the application, making the state capture all the more complex. For this reason, mechanisms for preempting arbitrary FPGA applications so that they can be reliably resumed later are not clear.

Instead, existing approaches to timeshare an FPGA avoid preemption [50] and rely on two techniques. The first is a "task-based" approach where work units are submitted to the FPGA and run to completion much like Ptasks [44]. Secondly, as a last resort a badly-behaved piece of user logic can simply be deconfigured by the OS, in a manner analogous to killing a misbehaving process. The scheduling problem then becomes one of dispatching tasks to the FPGA.

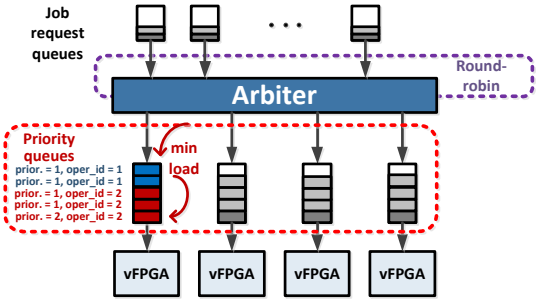


Figure 3: Coyote scheduling

The key *quantitative* difference with FPGA scheduling is that context switch time is much higher: reconfiguring a dynamic region can take many milliseconds. Moreover, only one region of the FPGA can be reconfigured at a time. If not addressed, these limitations can lead to unacceptably high scheduling overhead.

Coyote approach: Coyote adopts the task-based technique, with tasks being described by job objects. Tasks are not scheduled by the FPGA itself, instead the runtime manager on the host CPU schedules them spatially (across vFPGAs) and temporally (by reconfiguring a vFPGA if required, and serializing such reconfigurations).

The current version of Coyote adopts a modified priority-based queue scheme for tasks (Figure 3). Application software submits a task to a per-application queue in the runtime monitor. These are serviced in a round-robin fashion, and dispatched to a priority queue for one of the fixed number of vFPGA instances. Each of these queues is sorted first by priority and, then, by the bitstream image that the task requires.

This heuristic provides a degree of fairness between applications (though it could certainly be improved with better protection against starvation in a few pathological cases), but more importantly groups together tasks that can run in a sequence without intervening reconfigurations of their vFPGA. This approximates some of the benefits of Optimus [32], which employs a more static assignment of logic to vFPGAs but shares this between applications. Note that it also makes the scheduler non-work-conserving.

Discussion: For the 5 applications we evaluate in Section 4, the reduction in the number of required reconfigurations substantially improves efficiency, to the extent that, with current hardware, it probably dominates other aspects of the scheduling algorithm. However, we feel there is still important work

to be done in improving fairness, starvation-freedom, and predictability of the scheduler.

Coyote deliberately avoids any question of preempting applications running in vFPGAs, except *in extremis* to "kill" badly behaving user logic. This decision is worth discussing in more detail, since other approaches [26,32] provide explicit preemption interfaces. Applications can use these interfaces to implement user logic to save and restore their state in response to a preemption request from the scheduler.

The first reason for this decision is from an OS designer's perspective: the classical OS design principles adopted in Coyote strongly argue against this approach to preemption. Traditionally, user applications are not trusted to behave nicely by the OS, and so implementations take great care to ensure that preeemption never requires cooperation from the application – even in cases where it is explicitly visible to user threads, as in Psyche [33]. So-called "cooperative multi-tasking systems" (for example, early versions of the Apple Macintosh OS) do require application cooperation for context switching, but are generally not preemptive and, as history shows, are invariably supplanted by preemptive scheduling that does not require participation by the application.

The second reason is that the nature of "services" (e.g. networking) on an FPGA is different from that on a CPU. FPGAs emphasize spatial multiplexing and extreme concurrency. This means that services like the network stack (Section 3.6) and physical memory management (Section 3.5) do not need to be scheduled in Coyote: they are separate circuits and so inherently run all the time. A user-supplied preemption implementation may appear sufficient where these OS facilities are absent, but their presence means that user-implemented preemption has to also save and restore state (such as network flows) in each of these services. This capture of system-wide state cannot yet be done efficiently in current FPGAs.

3.4 Virtual memory

In a conventional OS, virtual memory provides a potentially unlimited number of "virtual address spaces" to software processes. By default, a virtual address space provides a sandbox of private memory, but segments of memory can be selectively shared between address spaces by the OS.

Virtual address spaces solve several crucial problems in computer systems: code and data does not need to be relocated at runtime, since it can be compiled and linked to run at a fixed address. Demand paging to a disk or SSD allows the amount of memory seemingly available to all applications to exceed the total real memory in the system.

Fragmentation of physical memory is avoided at anything coarser than page granularity. Physical locations for data can be chosen carefully to provide cache-coloring transparently to user code. Accesses to memory regions can be tracked via a "protect-and-trap" technique, with applications ranging from garbage collection [4], copy-on-write, and transaction management [35] to dynamic binary translation [10].

Hardware support for the abstraction of virtual memory is traditionally provided by the MMU, by way of three key functions: *address translation* from a virtual to a physical address space, *protection* of memory pages, and a mechanism to *trap* to the OS on certain memory accesses (i.e. a page or protection fault).

What's different on an FPGA? Some uses of virtual memory do not make sense on an FPGA, such as trapping on particular instructions or memory addresses. However, others (demand paging, relocation, etc.) are highly relevant.

Existing approaches to programming FPGAs generally ignore virtual memory, or handle address translation solely in the host OS kernel [11,14,17,26,27,39,62,63]. Pinned physical buffers are allocated and shared between FPGA user logic and software, which (when the data is not simply copied *en masse* between host memory and the FPGA) entails either the use of offsets to implement pointer-rich data structures, or cumbersome "pointer swizzling" when passing ownership of regions between devices. In both cases, one cannot simply pass a pointer from software to user logic without some mediation, typically by the OS kernel, or a runtime specific to a programming model like OpenCL [55].

One approach to accessing host virtual memory from the FPGA is via the host platform's IOMMU. However, IOMMUs are not well-suited to a dynamic set of FPGA applications, even the subset that only use PCIe as an interconnect. Optimus [32] has a good explanation of the limitations of IOMMUs, and employs an ingenious "page table slicing" technique to work around them. Other recent work also implements some form of translation on the FPGA from user logic to host-physical addresses [6,15,42,52], .

These approaches, however, are limited to one special case: user logic accessing data on the host CPU memory in a software virtual address space. Modern FPGA platforms, however, have additional extensive memory closely coupled to the chip (for example, Enzian's FPGA has 512 GiB of DDR4), and also devices (such as network or storage controllers). To interact correctly with other OS abstractions (such as device virtualization, isolation, or even simply access to FPGA resources from the CPU), a virtual memory abstraction for multi-tenant FPGAs must thus be extended to these resources as well. It must enable safe and securely access to memory both on the FPGA itself, and on memory and devices directly attached to the FPGA, from user logic *and* software.

An approach satisfying these requirements is present in modern GPUs [38] where a unified memory abstraction of GPU and host memory is implemented. This abstraction provides primarily increased programmability which removes the need for explicit memory management and data movement from the software side.

FPGAs also differ fundamentally from GPUs or other accelerators in that they are reconfigurable. In a CPU or, indeed, a conventional IOMMU or SMMU, parameters such as TLB

size, associativity, coverage, etc. are fixed when the chip is laid out. They represent a careful, "one-size-fits-all" compromise intended to run most workloads reasonably well.

In contrast, with an FPGA TLB parameters can be changed on the fly to handle specific workloads more efficiently. Moreover, many accelerator workloads which deal with large data volumes benefit greatly from larger page sizes – this motivates Optimus [32] to use 2MiB pages exclusively, for example. These factors, combined with the lower clock speed at which FPGAs run (typically about 10% of CPU clock speed), make a *software loaded TLB* more attractive as a design option.

Coyote approach: Rather than relying on a single shared FPGA MMU, and/or relying on an IOMMU, Coyote includes TLBs in the wrapper of each vFPGA. This not only allows TLB dimensions to be decided based on the application, but also provides sandboxing of user logic regardless of whether it is accessing off-chip DRAM attached to the FPGA's memory controllers, or host CPU DRAM using the xDMA engines in the static region. It also makes floorplanning and routing easier on the chip by reducing fanout.

Moreover, the TLBs are positioned in each dynamic region so as to mediate *all* accesses to FPGA-attached devices and RAM, and the entire host CPU's physical address space, something not possible with a conventional IOMMU.

The operation of TLBs in Coyote is best described in two parts: first, the underlying mechanism, and second, the different memory usage models it supports.

Mechanism: Coyote actually provides two TLBs per vFPGA, one for 4KiB pages and one for 2MiB large pages. The TLBs are fairly straightforward caches (see Figure 4); associativity and number of sets are determined at build time for the application. A TLB miss causes an interrupt to the host CPU, whereupon the driver identifies the faulting vFPGA and either loads the TLB with a valid mapping or signals a page fault, which would be handled in software on the host.

All accesses to both FPGA and host DRAM *from user logic* use the same unified TLB interface. User logic can therefore access any host memory, if the TLB allows. Accesses to host and FPGA memory use different paths to proceed in parallel.

Meanwhile, on the *host* side, the CPU's physical address space contains a region for each vFPGA, each of which is further subdivided into three parts:

1. The TLB contents and other privileged configuration values. This subregion may only be mapped by the privileged Coyote device driver.

2. Dynamic wrapper registers accessible to user software, e.g. for setting up DMA copies.

3. Direct access to user logic. CPU-initiated accesses are presented to user logic as AXI4 transactions (see Figure 2) to be interpreted as the user logic sees fit.

Usage models: The most common way of using this facility is to provide GPU-style "Unified Memory", which is

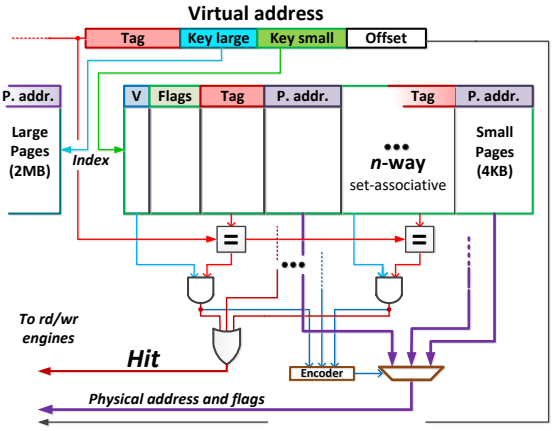


Figure 4: Coyote per-application TLBs

essentially a form of local distributed shared virtual memory: pages are copied (faulted in) on demand between FPGA and host memory via DMA with coherence managed by a combination of driver software and dedicated "page fault" units in the secure wrappers. Coyote can thus handle multiple application contexts maintaining different shared virtual address spaces. It is the job of the software component of Coyote to ensure that address mappings are consistent between the vFPGA TLBs and the virtual address space of the corresponding software processes. Though there is no fundamental need for them to be so, it allows direct sharing of pointer-rich data structures between application software and user logic.

Alternatively, TLB entries can indicate that, when a corresponding virtual address is requested, the physical access is directly routed to either host or FPGA memory without any copying of pages. For efficient random access (such as pointer-chasing) this may be much more faster to program and execute than "unified memory".

Finally, it is also possible to route CPU accesses to addresses on the FPGA back through the vFPGA wrapper's TLBs and into FPGA (or, indeed, host) memory. While quite slow on PCIe-based systems, it might be an attractive option on a fully-coherent non-PCIe system like Enzian.

Discussion: As we show in Section 4, the TLBs impose very little space overhead in a vFPGA, and deliver in return considerable simplicity in programming applications.

The partitioning of TLB functionality across vFPGAs brings a degree of performance isolation to vFPGA applications: one vFPGA cannot pollute the TLB contents of another region and thereby impact performance, an important consideration for a multi-tenant environment.

Note also that the area occupied by TLBs can be traded off against performance in an application-specific manner. This would not be possible with conventional IOMMUs situated on the PCIe bus, and would be hard to achieve with a

single IOMMU shared between applications. Partly as a consequence, we have yet to see serious performance overheads due to the software-loaded TLBs.

3.5 Memory management

In addition to virtual memory, a traditional OS provides facilities for managing *physical* memory. As hardware has become more complex, this has become more important for performance. E.g. an application might request regions of contiguous RAM to optimize sequential access and/or TLB coverage via superpages, or memory on specific NUMA nodes, explicitly (e.g. via `libnuma` on Linux) or implicitly (e.g. using Linux' "first-touch" allocation policy).

These abstractions more concern performance than correctness. However, in the case of peripherals and heterogeneous accelerators it may be a requirement for software to work. A CUDA application may need to allocate GDDR memory on a GPU which is accessible (over PCIe) to CPU code. Alternatively, a device might only be able to DMA to and from a subset of the CPU-attached physical RAM.

In a software OS, however, there is a *single* mechanism available to allocate memory with the right characteristics: choosing an appropriate range of physical addresses. The physical address functions as a proxy for all kinds of features of the hardware interconnect, memory controllers, DMA capabilities, and (in the case of cache coloring) the processor's cache architecture and placement policies.

What's different on an FPGA? Since FPGAs are much closer to the hardware, the situation is very different. Code running on FPGAs can access memory controllers directly. Data paths are not limited in size to cache lines, machine words, or MMU pages. SDAccel [58] exposes memory controllers explicitly to the programmer, providing flexibility but sacrificing simplicity and portability across FPGA devices.

The memory potentially visible to FPGA user logic is much more diverse than in software (and there are no caches). Block RAM (BRAM) is fast but scarce, DRAM is slower but there is typically much more of it, many systems have extensive off-chip DRAM available, and newer FPGAs incorporate High-Bandwidth Memory (HBM) as well.

Moreover, as with servicing TLB misses, it may be useful to offload the dynamic allocation of FPGA memory to software, although hardware allocators have been developed [61].

Coyote approach: Allocation of physical memory both within and between vFPGAs in Coyote is handled by software, in the kernel driver, which also takes care of creating virtual memory mappings both for the user logic and application CPU code. A variety of physical memory types can be used (off-chip DRAM, host DRAM, HBM, etc.).

Accessing memory is similarly different. Whereas software deals with register loads and stores or cacheline fills and write-backs, *any* memory access in FPGA user logic is inherently, and explicitly, a copy operation from one location to another.

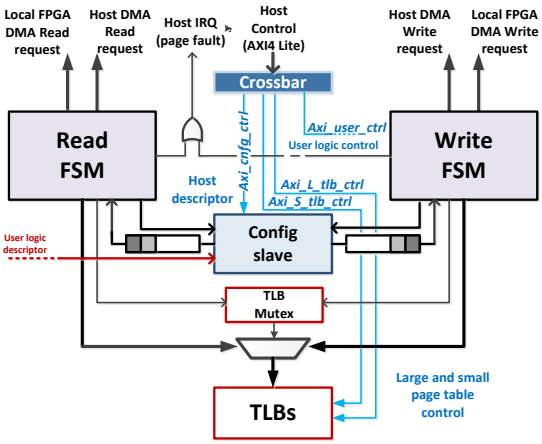


Figure 5: Read/Write engine

On current PCIe-based systems, user logic can access the entire host CPU's physical address space (albeit subject to memory protection) by transparently using xDMA copy engines. The complexity of routing memory accesses originating from both user logic and host software, and destined for host memory or FPGA resources, is handled by a *read/write engine* in each vFPGA wrapper (Figure 5), which provides this flexibility in access. Requests are submitted to the read/write engine using base/length descriptors; accesses from host software to FPGA memory are translated into these descriptors by the interface logic whereas the user logic issues them directly. This results in low overhead operations in the ULI entirely on virtual addresses. On fully coherent systems like Enzian, the read/write engines would be replaced with the interface Coyote provides to the CPU's native cache-coherence protocol.

In contrast to approaches like SDAccel (but more in line with a software environment), Coyote hides the presence of individual on-board DRAM controllers from user logic. On-board DRAM is the most commonly used way to hold bulk data in most FPGA acceleration algorithms, since it is higher capacity than BRAM. Coyote aims simply to maximize the bandwidth of bulk sequential access to this resource for user logic running in a vFPGA.

Coyote *stripes* DRAM access across all available controllers via careful allocation of pages. Coupled DRAM is allocated in 2MiB superpages. Each page is then striped across channels – e.g. if the FPGA has two physical DRAM channels, the first 1MiB of each page will access one DRAM channel, and the second half will use the second channel. This permits bandwidth optimization when performing rapid accesses with multiple channels present, and (as we show in Section 4.5) results in considerable performance gains over the naive approach. Accesses from different vFPGAs are still interleaved at each memory controller, as shown in Figure 6.

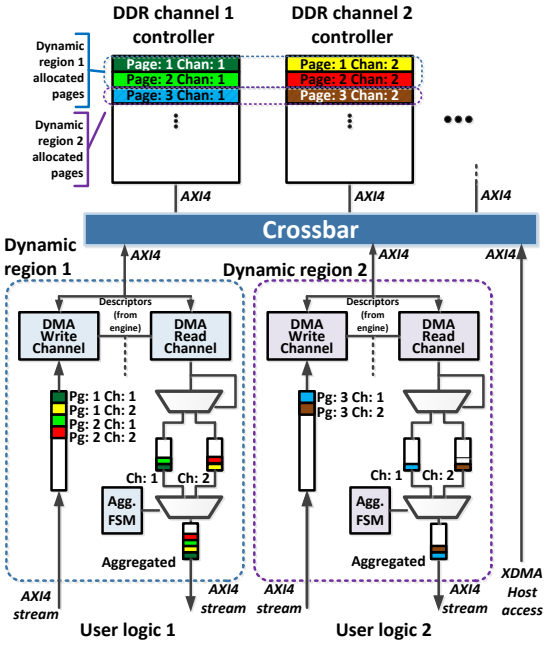


Figure 6: Multi-channel striping

Discussion: User logic is rarely as "memory-allocation intensive" as software, and so the kernel memory allocation code is rarely on the critical path.

By abstracting away on-chip DRAM controllers, Coyote makes a tradeoff in favour of portability and ease of programming, which we argue (based on our experiments) is appropriate. Moreover, Coyote applications can directly run on future FPGA designs which entirely offload memory controllers to dedicated hardware (as we discussed in Section 3.2).

In this context, striping provides more than just faster sequential access: it is vital for abstracting and sharing memory controllers since it allows the DRAM controllers to enforce fair sharing of bandwidth between vFPGAs.

3.6 IPC, I/O, and other services

A traditional OS provides a number of abstractions beyond those we have covered here. The most fundamental, at the heart even of microkernel architectures, is inter-process communication (IPC). We have already described how Coyote provides communication between vFPGAs and CPU-based software processes, but it also allows optional hardware queues between vFPGAs by analogy with IPC channels, pipes, etc., in a manner reminiscent of Centaur [42]. This allows users e.g. to chain dataflow operators running in different vFPGAs together while preserving the isolation between them.

While inter-vFPGA queues (and shared locations in FPGA virtual memory) can be used for inter-application communication, we find they are rarely used as such. As with containers, in our experience inter-vFPGA communication, when it happens at all, is coarse-grained and benefits from being independent of whether the vFPGAs share the same FPGA.

Instead, the main use for such queues is communication with *services* provided by Coyote.

For example, Coyote provides an optional, but fully integrated, high-performance *multi-tenant* network stack based on our open-source TCP/IP and RDMA engine for FPGAs [46,47]. Like the memory stack described in Section 3.5, the network stack abstracts away the details of the physical network interfaces present and exposes a portable, standard interface, and can be shared between all vFPGAs present.

Further services can be similarly implemented and configured into the static region at startup, for example a storage stack (perhaps driving directly attached Flash memory). The microkernel analogy applies here: Coyote provides a basic framework where such services can be added in the future based on use-case requirements.

Unlike in a software-based OS, Coyote "services" like the network stacks do not need to be scheduled, since they are always present on the FPGA – the FPGA is being spatially rather than temporally shared between services and user logic.

4 Evaluation

We focus on the question of whether the qualitative benefits of using Coyote's OS-style abstractions (scheduling, virtual memory, etc.) incur an acceptable quantitative cost in performance or efficiency. We look at overhead and space costs, as well as fairness in sharing resources, and the benefits of some of the optimizations in Section 3.

The hardware used for the results we report on here is a Xilinx VCU118 board [56] with an Ultrascale+ VU9P FPGA, attached to a host PC via PCIe x16. This interface provides a maximum theoretical bidirectional bandwidth of 16GiB/s. The board has 2 external DDR4 banks connected to the FPGA. Each DDR channel has a soft core DRAM controller instantiated in the FPGA fabric providing a total theoretical bandwidth of 18GiB/s. The host PC is a quad-core Intel i5-4590 at 3.3 GHz with 8GiB of RAM running at 1600MHz.

Unless stated otherwise, the system frequency used on the FPGA is 250 MHz. While each Coyote vFPGA has a separate PLL-generated clock, all the experiments reported here used the same frequency for the vFPGAs.

4.1 Macro-benchmark: decision trees

We first compare the performance of a complete, mature application running on Coyote with that obtained on Amazon F1 instances with the Xilinx SDAccel programming framework [58] and the Intel HARP environment [39]. It is hard to draw detailed conclusions from such a coarse-grained comparison, but we aim to show that (1) Coyote is a viable platform

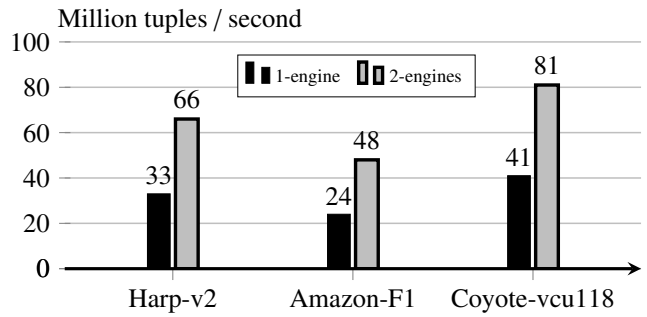


Figure 7: Performance of decision trees.

for real applications, and (2) the portability and programming features of Coyote come with negligible performance cost. The application is an open-source implementation [40, 41] of Gradient Boosting Decision Trees [34], focussing here exclusively on inference over decision tree ensembles.

Decision trees are a popular form of supervised machine learning for widely used tasks like classification and regression. They are constructed by recursively splitting the data into multiple groups. Using a cost function splits are evaluated and a greedy algorithm decides which split is the best candidate. Recursive splitting terminates at a pre-determined tree depth to prevent overfitting.

To do inference on the FPGA, the tree model is first loaded into FPGA on-chip memory. Data is then fetched from the host, inference performed, and the results copied back to host memory. All three phases are overlapped, allowing computation latency to be hidden behind memory operations.

We compare inference throughput (scored tuples per second) over Coyote with the same application on F1, and with a port running on Intel's hybrid CPU-FPGA HARP v2 platform. The latter is a rather different platform and the FPGA is clocked at 200MHz instead of 250MHz. Since F1 targets OpenCL applications, the SDAccel port employs a strict GPU-based compute model which incurs high data transfer overhead. In all cases, we measure throughput with both one and two instances of the application running on the FPGA with the data size of 4k tuples. On all platforms, the inference engine is compute-bound and requires only 4 GiB/s of memory bandwidth, allowing two instances to operate at full capacity.

The results are shown in Figure 7. Coyote provides comparable or better performance to that of the two commercial baselines. In the case of F1, this is despite Coyote providing portability and supporting multiple vFPGAs (SDAccel only allows a single dynamic region on the FPGA). True comparison with HARP is more tentative, given the lower clock frequency and very different hardware.

Nevertheless, we can conclude that, at the very least, there is no performance penalty in using Coyote in this case, and benefiting from the qualitative value it brings.

# vFPGAs	Stacks	LUTs	BRAM	Regs
1	✗	4%	4%	2%
2	✗	5%	5%	3%
4	✗	6%	7%	4%
1	✓	9%	10%	6%
2	✓	11%	12%	7%
4	✓	14%	14%	9%

Table 1: Resource overhead

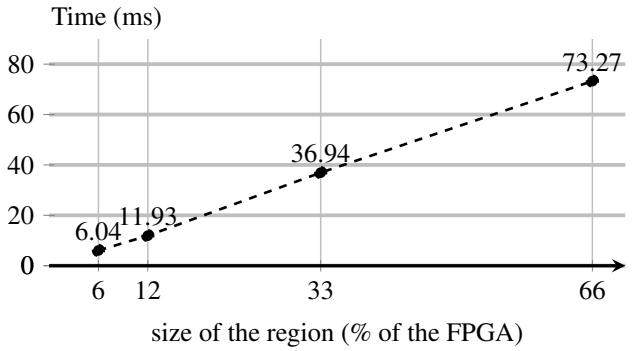


Figure 8: Time to load the operator and provide the view.

4.2 Space overhead

Raw performance is not the only consideration when comparing FPGA implementations, however. The space overhead (more precisely, the various resources on the chip used for the framework) can be just as important.

In this regard, Coyote (or any such set of abstractions) is strictly worse than a custom, native implementation of an application that takes over the whole FPGA, just as a bare-metal program is likely to use fewer resources than one running on top of Linux or Windows.

The space overhead of the framework for varying numbers of virtual FPGA regions and configurations with and without memory and network stacks are shown in Table 1. We give figures for the principal logic resources on the FPGA: lookup tables (LUTs), block RAM (BRAM), and registers (Regs).

On the VU9P, Coyote incurs a base overhead of 2-4%, increasing by $< 1\%$ for each additional vFPGA. Adding network and memory stacks roughly doubles this, incurring at most 14% for a full-featured Coyote install with 4 vFPGAs.

Larger future FPGAs and migration of often-used functionality into hard IP are likely to reduce this overhead still further. We therefore consider this to be a modest penalty in return for the benefits Coyote offers.

4.3 Micro-benchmark: context switching

We next measure the performance penalty in context switching a vFPGA from one application to another.

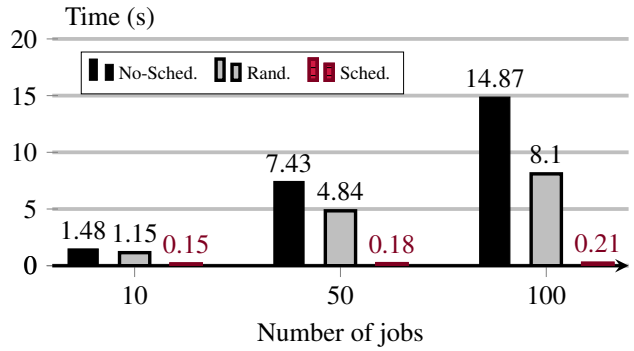


Figure 9: Scheduling performance.

Partial reconfiguration is still relatively slow on FPGAs, despite recent improvements. Figure 8 shows partial reconfiguration latency of the Xilinx VU9P as a function of the chip area to be reconfigured; Coyote with 4 vFPGAs would correspond to roughly 20%, or 20ms, per vFPGA.

We note that achieving even this performance is non-trivial. Coyote's implementation uses a fast data stream to the ICAP (the FPGA unit handling partial reconfiguration) which can saturate its bandwidth of about 800MiB/s. In contrast, SDAccel achieves a mere 19MiB/s over a slow AXI4-Lite link.

As mentioned in Section 3.3, this overhead can be reduced substantially by sharing the same vFPGA logic between successive tasks targeting the same partial bitstream (compute operator). We illustrate this with a simple example by scheduling queues of tasks from the four applications used in Section 4.4, all with small transfers of 4KiB each and all with the same priority. We configure Coyote here with 3 vFPGAs.

We dispatch jobs in three ways: `no-sched` is round-robin in both queues and vFPGAs. This causes a reconfiguration for each job and is the worst-case scenario. `rand` picks the next job from a random queue each time, and so has a 1-in-3 chance of needed a reconfiguration, and `sched` uses Coyote's heuristic of grouping jobs which share the same user logic.

Figure 9 shows total turnaround time for 10, 50, and 100 jobs of each type. Unsurprisingly, minimizing the number of partial reconfigurations has a dominating effect on total system throughput. Clearly there is room here for much more sophisticated job scheduling, beyond the scope of this paper.

4.4 Resource sharing

We now evaluate how the OS-like features of Coyote can provide fair sharing of resources across multiple vFPGAs simultaneously. In cloud deployments, stable and predictable distribution of resources is a key requirement.

We run four different applications on Coyote: AES encryption, sha256 hash computation, HyperLogLog multiset cardinality estimation [18, 28] and k-means calculation [22]. In each experiment we run an application with one or more simultaneous vFPGA instances at a time. All tests except the k-means are using the host memory and direct streaming. Due to the iterative nature, the k-means utilizes accesses to the local FPGA memory.

We measure per-application round-trip throughput vs. transfer size, including the transfer of the plain data to the FPGA user logic, pipeline computation, and simultaneous transfer of the computed results back to the memory. We use hardware counters in the FPGA fabric and so incur no overhead.

When multiple applications are running concurrently, we also calculate mean absolute deviation (MAD) of the instances from the average performance results, to give a quantitative measure of (un)fairness in resource allocation.

Results are shown in Figure 10. sha256 (Figure 10.a) is compute bound, and performance scales perfectly as long as all the vFPGAs fit in the FPGA.

More interesting is AES (Figure 10.b) which is memory-bound. Here multiple AES vFPGAs are competing for PCIe bandwidth, which is saturated in all cases due to the AES implementation being heavily pipelined. We observe that, firstly, throughput of an AES instance is inversely proportional to the number of peers in the system, showing that the scarce resource of PCIe bandwidth is being shared between them, and also the MAD is very low compared with total bandwidth, suggesting that sharing is fair.

The HyperLogLog implementation uses 16 parallel pipelines that are able to compute on a single cache-line at a time. The module is thus able to sustain processing at line rate for larger transfers (as are mostly present during cardinality estimation). For smaller transfers processing latency is the domineering factor. The results are shown in the Figure 10.c.

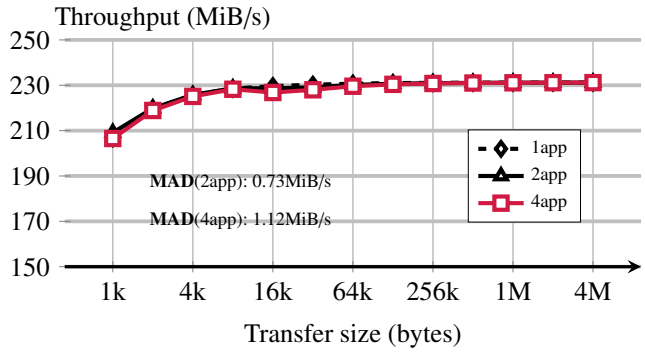
The final results (Figure 10.d) show the throughput of the k-means clustering operator during the single computation iteration. This is an iterative algorithm, where data is first offloaded from the host to the local FPGA memory. The data is then streamed to the user logic in each iteration and dispatched to 16 parallel pipelines on the FPGA to compute centroids, the results of which are then transferred back.

In summary, these results validate our goals of sharing scarce bandwidth on the FPGA between multiple tenants.

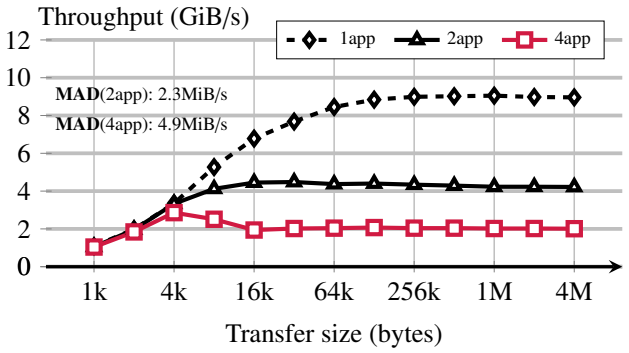
4.5 Striping

We evaluate the impact of Coyote hiding individual DRAM channels behind a single abstraction that stripes each 2MiB page across all channels on the FPGA for portability.

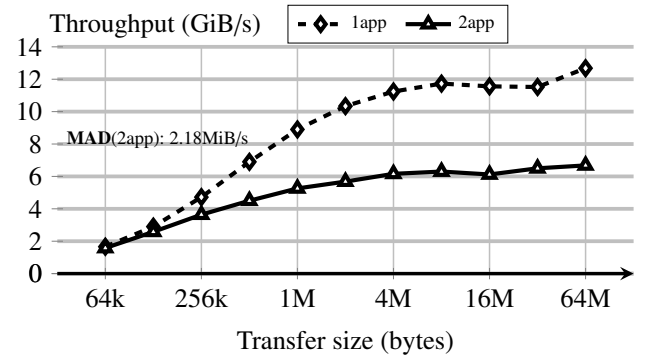
The benchmark is a simple DRAM to DRAM copy, implemented entirely on the FPGA and measuring throughput for transfers ranging from 4KiB to 1MiB. We measure bandwidth for three scenarios. First, `1-channel` copies memory using on a single channel, and is the baseline for performance. Second, `2-channel` reads from one channel and writes to another. This is the best case and requires knowledge of all the channels in the FPGA. It could be achieved in, for example,



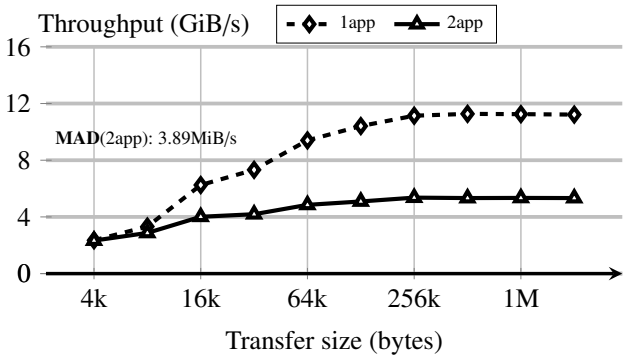
(a) sha256 round throughput for different number of concurrent applications in dynamic regions.



(b) AES round throughput for different number of concurrent applications in dynamic regions.



(c) HyperLogLog throughput.



(d) k-means single iteration throughput.

Figure 10: Performance benchmarks for example applications running in Coyote showing fair sharing of the bandwidth.

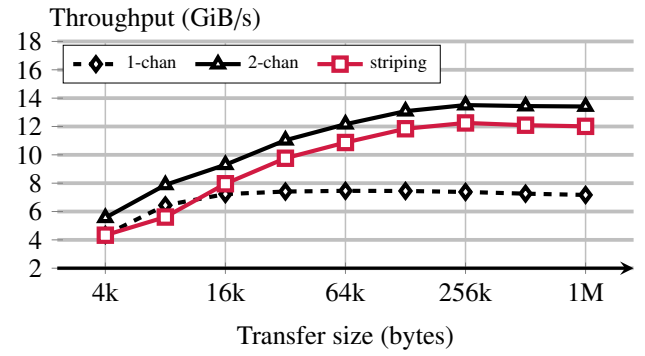


Figure 11: Striping performance.

SDAccel by explicit placement of data on the channels and careful FPGA-specific code. Finally, `striping` shows Coyote's performance when oblivious to channels and placement, and each data page is striped across channels.

Figure 11 shows the results. For small transfers, setup costs dominate, but a single channel becomes saturated at 16KiB and two channels at about 128KiB. Coyote incurs an overhead of about 10%, which is competitive with many cases of hand-optimized vs. compiler generated software code, and leads us to conclude that abstracting the DRAM controllers is a worthwhile trade-off for performance isolation and portability.

4.6 Demand paging

GPU-style "unified memory" implements a form of distributed shared virtual memory between the host and FPGA, largely abstracting memory management and explicit data movement from the users. When a vFPGAs tries to access virtual locations on the local FPGA memory which are not present in the physical memory, a page fault is generated and the driver initiates a copy from host to FPGA memory, then adjusts page tables on both sides, before signalling the vFPGA that it can proceed.

Without this model, explicit copying of data would be required, as illustrated by this pseudocode:

```
void* host_d = malloc(size);
void* fpga_d = getFpgaMem(size);
offloadMemCpy(host_d, fpga_d, size);
executeOperator(fpga_d, size);
free(host_d);
freeFpgaMem(fpga_d);
```

The demand paging provided by "unified memory" allows a simpler (for the programmer) model, resulting in code like this:

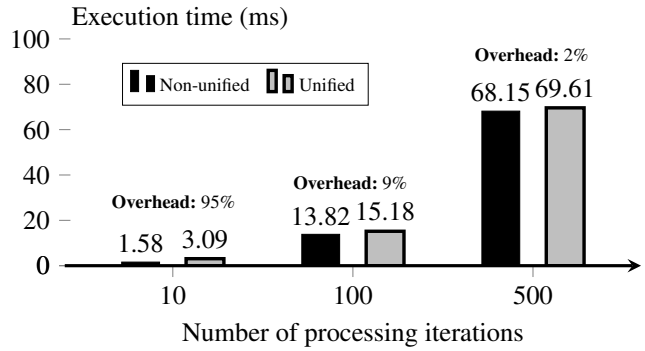


Figure 12: Unified memory overhead.

```
void* host_d = malloc(size);
executeOperator(host_d, size);
free(host_d);
```

In a fully cache-coherent system like Enzian [21] this code would not require copies at all, but in PCIe-based systems they are needed in both directions: after the computation has completed, the pages holding the computation results must also be copied back to the host physical memory.

The cost, therefore, of the unified memory abstraction stems from page faults on both sides, remapping pages by modifying page tables, and copying the data between host and FPGA. In practice, this cost is amortized over the number of iterations (for example) that the computation performs for each transfer.

Figure 12 shows this overhead in the context of the whole computation. The workload represents kmeans iterating over 1MiB of data which has to be moved from the host to local FPGA memory. We vary the number of iterations and measure the impact of the page fault overhead and initial copy.

For a single iteration the overhead is high (95%), reflecting the fact that an iteration executes extremely quickly on the FPGA and is comparable in time to the copy, and suggesting that this model of memory usage is not ideal for streaming applications. However, as the iteration count reaches 500, the overhead has reduced to 2% and is likely to be an acceptable price to pay for programming convenience.

5 Related work

The FPGA community has generated a tremendous amount of work in recent years on programming and managing FPGAs. We have compared Coyote with many examples already in Section 3, and two recent surveys [24, 50] give an excellent overview. In this section, therefore, we focus on a few important recent systems.

The initial version of Microsoft's Catapult [12, 43] environment offers a reusable, static portion of programmable logic accessible through a high level API. Configurability for the (single) application is possible for modules like the network (recently offloaded to a sophisticated ASIC) and memory. The accelerator/smartNIC usage model means there is no support for virtualization nor partial reconfiguration.

Intel's hybrid CPU-FPGA HARP design [39] turns the FPGA into one more processor. Intel implements its own QuickPath interconnect [23] supporting full cache-coherent memory access, but only to external memory on the CPU side. Usage model and management is similar to Catapult. Partial reconfiguration is possible but there is no option to include local on-board memory or network modules on the FPGA.

Xilinx SDAccel [58], used by Amazon [3] and Alibaba [1] in their cloud deployments, also divides the FPGA into a static "shell" and dynamic user regions. One application can run at a time, but the user logic can be exchanged at run time with the help of partial reconfiguration. To date, there is no support for I/O devices or network.

All these deployed systems support only a single application at a time, and also do not try to provide a shared virtual address space between host software and user logic. Systems in the research literature are rather more ambitious in adopting one or more ideas from traditional operating systems:

AmorphOS [26] aims to increase FPGA utilization by placing multiple applications on the FPGA. It provides protection on FPGA-attached memory, but no access to host memory. Protection is based on segments set up by the host OS. AmorphOS can operate in "low-latency mode", where applications occupy different parts of the dynamic region, and "high-throughput" mode, where everything is synthesized into a single bitstream. Time-division multiplexing in low-latency mode is achieved by requiring applications to implement correct checkpoint and resume.

AmorphOS can be seen as pushing many traditional OS problems into the synthesis pipeline, and compiling many different bitstreams for configurations which, in Coyote, are handled at runtime by the same image. Since it provides no integration with the host memory system, and applications are directly compiled to the FPGA, AmorphOS provides no virtual memory facilities beyond segmented addressing of FPGA memory. Scheduling is simplified by not partially reconfiguring the FPGA, which also obviates the need to provide a uniform network interface.

AmorphOS optimizes how many applications can fit on one FPGA, at the cost of compilation and deployment overheads, by delegating OS functionality to synthesis tools. In contrast, Coyote's OS-centric approach standardizes the execution environment, allowing applications to be flexibly deployed, and evaluates the cost of this generality.

Optimus [32] provides FPGA user logic with access to host memory via a per-application virtual address space. It partitions the dynamic region into application containers which appear not to be partially reconfigurable, but which can run the same user logic on behalf of multiple applications. As in AmorphOS, user logic implements checkpoint and restore to allow time-division multiplexing of resources. Optimus allows the host address space to be shared, but does not give

a host process access to the address space of a vFPGA.

Optimus has many similarities with Coyote, but focuses on a subset of OS functionality. By avoiding dynamically reconfiguring vFPGAs, scheduling is simplified relative to Coyote. Optimus provides address translation, but only for vFPGA-to-host access, whereas Coyote provides a true unified virtual address space shared between host process and user logic. This in turn allows Coyote to virtualize services like the network stack, something Optimus does not do. Optimus therefore does not provide a standard execution environment for bitstreams, since the functionality it does provide would not benefit from such an environment.

ViTAL [62] focusses on clusters of FPGAs and, unlike Coyote, addresses distributing applications across a cluster. While it provides a network device, and flexible multiplexing, it does not target hybrid CPU/FPGA systems, and provides neither unified memory, nor (e.g.) a shared network *stack* between vFPGAs.

ViTAL virtualizes access to the FPGA memory (like AmorphOS) and the network *device* (as a simple point-to-point communication link). As with Coyote, it uses a fixed partition of the dynamic region in reusable vFPGA which are allocated to applications when they are deployed. A key feature of ViTAL is being able to partition applications and compile them into multiple vFPGAs; these can then be deployed on the same FPGA or several connected by point-to-point links.

By not supporting host memory access nor virtualizing a high-level service like TCP or RDMA, ViTAL is relieved of the need for a virtual memory system. Moreover, by using the compiler to turning a set of physical FPGAs into one large logical FPGA by application partitioning, it obviates the need for a standard execution environment.

To greater or lesser degrees, all these systems focus on optimizing one or another metric and implement a subset of the critical functionality of a classical OS.

In contrast, Coyote investigates the consequences of a complete, general-purpose approach: putting a general OS feature set together (multi-user TCP/IP stack, unified memory translation/protection across CPU and FPGA, inter-application communication, standardized execution environment, etc.). Uniquely, this combination is what allows Coyote to provide shared high-level OS services like networking.

It also demonstrates that a full set of combined OS features fundamentally changes how a system like Coyote is designed, and this is where it differs most from prior work while still reusing a number of ideas from such systems. We return to this point in our conclusion.

6 Conclusion

Coyote approaches the FPGA shell as an *operating system design problem*. While putting individual OS features on an FPGA has value, taking a holistic view allows us to identify how things fit together. The design of virtual memory on an FPGA changes radically when one takes into account e.g. FPGA-local devices, or the need to abstract local DRAM controllers. Conversely, abstracting such controllers only works when one has the right MMU design in place.

For example, allowing both software and hardware applications to initiate virtual memory accesses to both host and FPGA memory resources enables a uniform execution environment and portability across different memory systems, but may rule out the application-implemented checkpoint-and-restore approaches ViTAL and Optimus use for cooperative "preemption", since there is now per-application state (TLBs, etc.) not accessible to user logic.

As FPGAs become larger, the demand for the traditional OS functions of secure multiplexing, sharing, and abstraction will grow. At the same time, so will the opportunity to provide more OS-like functions on the FPGA. It is important that these functions work together. Our evaluation shows that the price of this complete OS functionality is more than acceptable in throughput, space efficiency, scheduling overhead, and memory bandwidth.

A further hardware trend, moreover, is the migration of commonly-used functions out of synthesized logic and into hard IP cores on the FPGA. In this rapidly-changing landscape, the right set of abstractions can prevent hard-to-develop OS-style logic from becoming rapidly obsolete.

Experience with software also suggests that OS abstractions are “sticky”: once decided, they alter very slowly over time even when the underlying hardware changes radically, to the detriment of performance and security. This suggests that it is vitally important to get things “right” as early as possible.

Coyote is a small step in this direction, and shows that a coherent and reasonably complete set of OS abstractions, suitably modified, can map well onto an FPGA, deliver both immediate and longer-term benefits, and impose only a modest overhead on today’s hardware.

Coyote can be downloaded at https://github.com/fpgasystems/Coyote.

Acknowledgements

This work has been made possible through a generous equipment donation from Xilinx and through access to the Xilinx Adaptive Compute Cluster (XACC) Program. The research of Dario Korolija is funded in part by a grant from HPE. We would like to thank Mohsen Ewaida, Zhenhao He and Amit Kulkarni for some of the use cases and designs, David Sidler for feedback and help with the network stack, the anonymous reviewers for their helpful comments and questions, and our shepherd Baris Kasikci for feedback and guidance.

References

[1] Alibaba Cloud Services. Compute optimized instance

families with FPGAs. https://www.alibabacloud.com/help/doc-detail/108504.htm, May 2020.

[2] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Owaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. Tackling Hardware/Software co-design from a database perspective. In *Proceedings of the 6th biennial Conference on Innovative Data Systems Research (CIDR)*, Amsterdam, Netherlands, January 2020.

[3] Amazon Web Services. Amazon EC2 F1 Instances: Enable faster FPGA accelerator development and deployment in the cloud. https://aws.amazon.com/ec2/instance-types/f1/, May 2020.

[4] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, page 96–107, New York, NY, USA, 1991. Association for Computing Machinery.

[5] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of FPGA, GPU and CPU in image processing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 126–131, Aug 2009.

[6] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A Fahmy, and Paolo Ienne. Virtualized execution runtime for FPGA accelerators in the cloud. *IEEE Access*, 5:1900–1910, 2017.

[7] David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *ACM Queue*, 11:40:40–40:52, 2013.

[8] Donald G. Bailey. The Advantages and Limitations of High Level Synthesis for FPGA Based Image Processing. In *Proceedings of the 9th International Conference on Distributed Smart Cameras*, pages 134–139. ACM, 2015.

[9] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A `fork()` in the road. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS-XVII)*, Bertinoro, Italy, May 2019.

[10] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing Virtualization to the X86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4), November 2012.

[11] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116. IEEE, 2014.

[12] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, October 2016.

[13] CCIX Consortium and others. Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com, January 2019.

[14] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 3. ACM, 2014.

[15] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: An In-fabric Memory Architecture for FPGA-based Computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106. ACM, 2011.

[16] CXL Consortium. Compute Express Link. https://www.computeexpresslink.org/, May 2020.

[17] Suhaib Fahmi, Kizheppatt Vipin, and Shanker Shreejith. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435. IEEE, 2015.

[18] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In Philippe Jacquet, editor, *AofA: Analysis of Algorithms*, volume DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) of *DMTCS Proceedings*, pages 137–156, Juan les Pins, France, June 2007. Discrete Mathematics and Theoretical Computer Science.

[19] D. Fortún, C. G. de la Cueva, J. Grajal, M. López-Vallejo, and C. L. Barrio. Performance-oriented Implementation of Hilbert Filters on FPGAs. In *2018 Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6, Nov 2018.

[20] B. Gottschall, T. Preußer, and A. Kumar. Reloc – An Open-Source Vivado Workflow for Generating Relocatable End-User Configuration Tiles. In *2018*

IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 211–211, April 2018.

[21] ETH Zurich Systems Group. Enzian, a research computer. http://enzian.systems, May 2020.

[22] Zhenhao He. Bit-Serial kmeans. https://github.com/fpgasystems/bit_serial_kmeans, October 2020.

[23] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, January 2009.

[24] C. Kachris and D. Soudris. A survey on reconfigurable accelerators for cloud computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, 2016.

[25] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 160–167, April 2017.

[26] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, 2018.

[27] Oliver Knodel, Paul R Genssler, and Rainer G Spallek. Migration of long-running Tasks between Reconfigurable Resources using Virtualization. *ACM SIGARCH Computer Architecture News*, 44(4):56–61, 2017.

[28] Amit Kulkarni, Monica Chiosa, Thomas Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. FPGA-based HyperLogLog Accelerator. https://github.com/fpgasystems/fpga-hyperloglog, October 2020.

[29] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, Feb 2007.

[30] X. Li, L. Ding, L. Wang, and F. Cao. FPGA accelerates deep residual learning for image recognition. In *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 837–840, Dec 2017.

[31] ARM Ltd. AMBA 4 AXI4-Stream Protocol. https://developer.arm.com/docs/ihi0051/latest, 2010.

[32] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 827–844, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. *SIGOPS Oper. Syst. Rev.*, 25(5):110–121, September 1991.

[34] Alexey Natekin and Alois Knoll. Gradient Boosting Machines, A Tutorial. *Frontiers in neurorobotics*, page 21, 2013.

[35] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 677–689, New York, NY, USA, 2015. Association for Computing Machinery.

[36] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70, June 2004.

[37] Eriko Nurvitadhi, Jeffrey J. Cook, Asit K. Mishra, Debbie Marr, Kevin Nealis, Philip Colangelo, Andrew C. Ling, Davor Capalija, Utku Aydonat, Aravind Dasu, and Sergey Y. Shumarayev. In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*, pages 106–110, 2018.

[38] NVIDIA Corporation. *Unified Memory in CUDA 6*, version: 2.0, document revision: 29 edition, Nov 2013. https://devblogs.nvidia.com/unified-memory-in-cuda-6/.

[39] Neal Oliver, Rahul R Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, et al. A reconfigurable computing system based on a cache-coherent fabric. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 80–85. IEEE, 2011.

[40] Muhsen Owaida and Gustavo Alonso. Distributed Inference over Decision Tree Ensembles. https://github.com/fpgasystems/Distributed-DecisionTrees, October 2020.

[41] Muhsen Owaida, Amit Kulkarni, and Gustavo Alonso. Distributed Inference over Decision Tree Ensembles on Clusters of FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 12(4), September 2019.

[42] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218. IEEE, 2017.

[43] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[44] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 233–248, New York, NY, USA, 2011. Association for Computing Machinery.

[45] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. https://arxiv.org/abs/1505.01120v1, 2015.

[46] D. Sidler, Z. István, and G. Alonso. Low-latency tcp/ip stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2016.

[47] David Sidler, Monica Chiosa, Zhenhao He, Mario Ruiz, Kimon Karras, and Lisa Liu. Scalable Network Stack supporting TCP/IP, RoCEv2, UDP/IP at 10-100Gbit/s. https://github.com/fpgasystems/fpga-network-stack.git, October 2020.

[48] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *EuroSys'20*, pages 29:1–29:16, 2020.

[49] J. Stuecheli, B. Blaner, C.R. Johns, and M.S. Siegel. CAPI: A coherent accelerator processor interface. *IBM J. Research and Development*, 59(1), 2015.

[50] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. A survey on FPGA virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–1317. IEEE, 2018.

[51] Malte Vesper, Dirk Kocha, and Khoa Phama. PCIeHLS: an OpenCL HLS framework. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, pages 10–15. IEEE, 2017.

[52] Pirmin Vogel, Andrea Marongiu, and Luca Benini. Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU. In *IEEE Transactions on Computers (Volume: 68 , Issue: 4 , April 1 2019)*, pages 510–525. IEEE, 2018.

[53] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. A Survey of FPGA Based Deep Learning Accelerators: Challenges and Opportunities. *CoRR*, abs/1901.04988, 2019.

[54] Skyler Windh, Xiaoyin Ma, Robert Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid Najjar. High-Level Language Tools for Reconfigurable Computing. In *Proceedings of the IEEE (Volume: 103 , Issue: 3 , March 2015)*, pages 390 – 408. IEEE, 2015.

[55] F. Winterstein and G. Constantinides. Pass a pointer: Exploring shared virtual memory abstractions in opencl tools for fpgas. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 104–111, 2017.

[56] Xilinx. VCU118 Evaluation Board User Guide. https://www.xilinx.com/support/documentation/boards_and_kits/vcu118/ug1224-vcu118-eval-bd.pdf, October 2018.

[57] Xilinx. DMA/Bridge Subsystem for PCI Express v4.1 Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf, November 2019.

[58] Xilinx. *SDAccel Environment User Guide*, v2019.1 edition, May 2019. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1023-sdaccel-user-guide.pdf.

[59] Xilinx. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*, v.1.3.1 edition, May 2020. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html#documentation.

[60] Xilinx. *Alveo U280 Data Center Accelerator Card Data Sheet*, v.1.3 edition, May 2020. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html#documentation.

[61] Zeping Xue and D. B. Thomas. SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, September 2015.

[62] Yue Zha and Jing Li. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 845–858, New York, NY, USA, 2020. Association for Computing Machinery.

[63] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 22. ACM, 2017.

A Artifact Appendix

A.1 Abstract

Coyote brings operating system abstractions to reconfigurable heterogeneous architectures. It provides a range of abstractions which ease the interaction between the host, the FPGA, the memory and the network. The following sections will describe the process of obtaining the framework resources and building them. The application deployment procedure is shown as well.

A.2 Artifact check-list

- **Compilation:** HLS, CMake, C++, Boost.
- **Run-time environment:** Vivado, Linux.
- **Hardware:** Xilinx.
- **Metrics:** Throughput, latency, resources, reconfiguration.
- **Experiments:** HyperLogLog, kmeans, AES, sha256, decision trees, microbenchmarks.
- **Required disk space:** 4MiB.
- **Expected experiment run time:** 2 hours.
- **Public link:** https://github.com/fpgasystems/Coyote.

A.3 Description

A.3.1 How to access

The open-source version of the Coyote framework can be found on Github at the following address:
https://github.com/fpgasystems/Coyote.

A.3.2 Hardware dependencies

The framework targets a variety of Xilinx data center and development boards. At this point full support for the following boards is provided: Alveo U250, Alveo U280, VCU118. The boards have to be attached to the host system over the PCIe. If available, the framework takes advantage of the AVX2 (*Advanced Vector Extenstions*) instruction set. Legacy support is also provided.

The hardware build process relies on the Vivado toolchain and its high-level synthesis extension. Versions 2019.2 and 2020.1 have been officially tested. The toolchain is used for the compilation of the full and partial bitstreams. It also handles the deployment of the full bitstreams. The automation of the hardware build process is done with CMake. Minimum required version is 3.0.

A.3.3 Software dependencies

The software build process is split between the low level Linux kernel driver and the high level user application layers.

The driver code was tested on the machine with the Linux kernel version 5.4. This code is built with Makefile.

The user application layer is fully written in C++11. The Boost libraries (https://www.boost.org/) are used for the parsing of the command line arguments. As with hardware, CMake with a minimum version of 3.0 is required for the software build automation.

A.4 Installation

Pull the newest version of the repository:

```
$ git clone https://github.com/fpgasystems/Coyote
$ cd Coyote
```

The network stack submodule and the correct branch can then be initialized:

```
$ git submodule update --init --recursive
```

At this point all the necessary resources are available locally. Further build is split into separate hardware and software processes.

A.4.1 Hardware build

This section explains the process of creating the custom hardware design, the integration of the arbitrary user logic and finally the formation of the valid FPGA bitstreams.

First create the build directory inside the hw directory:

```
$ cd hw
$ mkdir build
$ cd build
```

Enter a valid chosen system configuration:

```
$ cmake .. -DFDEV_NAME=u250 <params...>
```

Large level of configuration flexibility for the framework is available. This allows the framework to adapt to a variety of processing scenarios. The following parameters can be chosen (bolded parameters are passed by default):

- **FDEV_NAME:** This is the name of the target device. Supported parameters are <**u280**, u250, vcu118>.
- **N_REGIONS:** This is the number of concurrent vFPGAs (independent regions). The maximum of 16 regions per FPGA is supported at the moment <**1**:16>.
- **EN_STRM:** Enables the direct host-FPGA streaming over PCIe lanes <**0**, **1**>.
- **EN_DRAM:** Enables the local FPGA memory stack <**0**, 1>. It can work in conjunction with the streaming.
- **N_DRAM_CHAN:** The number of the chosen DRAM channels. The maximum available number depends on the target board. <**1**:4>.

- **EN_PR:** Enables the partial reconfiguration flow <**0**, 1>. This partitions the FPGA fabric into multiple dynamic regions. The number depends on the amount of vFPGAs present. A separate partial bitstream will be generated for each dynamic region. Manual floorplanning of dynamic regions is advised.
- **EN_TCP:** Enables the 100G TCP/IP stack <**0**, 1>. This integrates a TCP/IP network stack and exposes its communication interface to every vFPGA.
- **EN_RDMA:** Enables the 100G RDMA stack <**0**, 1>. This integrates a full RDMA network stack with reliable communication protocol (RC) built on top of RoCE v2. Interface is exposed to every vFPGA.

The build directory with the chosen configuration is initiated once the previous command completes. Now referenced high-level synthesis cores can be built:

```
$ make installip
```

The hardware project can then be created:

```
$ make shell
```

Once this command completes, the project with one static and initial vFPGA regions is created (config 0). If partial reconfiguration flow is enabled, additional sets of partial bitstreams (new logic for each vFPGA) can be created:

```
$ make dynamic
```

This command can be executed multiple times to create multiple sets of partial bitstreams (config 1, 2, 3, ...).

At this point the user logic can be inserted into vFPGAs. Wrappers can be found under build project directory in the hdl/config_X. Once the user design is ready to be compiled, run the following command:

```
$ make compile
```

When the compilation finishes, the initial bitstream with the static region can be loaded to the FPGA via JTAG. This can be done in Vivado's programming utility. At any point during the compilation, the status can be checked by opening the project in Vivado (`start_gui` command).

All compiled bitstreams, including partial ones, can be found in the build directory under bitstreams.

A.4.2 Driver

The driver can be compiled on the host machine:

```
$ cd driver
$ make
```

Once the bitstream is loaded on to the target FPGA, the rescan of the PCIe can be executed with the utility script:

```
$ ./util/hot_reset.sh
```

If during this card detection fails, warm reboot of the host machine has to be completed. The driver can then be inserted into the kernel:

```
$ insmod fpga_drv.ko
```

The software applications can now be executed.

A.4.3 Software build

This section explains the process of building the user applications that utilize the provided high-level API. Additionally, the scheduling example provides the runtime manager which abstracts the application deployment.

First create the build directory inside the directory of the chosen software project:

```
$ cd sw/<project>
$ mkdir build
$ cd build
```

Initiate the build configuration and compile the executable:

```
$ cmake ..
$ make main
```

System permissions need to be assigned to the executable.

A.4.4 Simulation

The user logic hardware can be simulated in Vivado:

```
$ cd hw/sim/scripts/sim
$ vivado -mode tcl -source tb.tcl
```

At this point any user logic can be inserted and arbitrary stimulus applied. The signal behaviour can then be observed.

A.5 Evaluation and expected result

The user logic for hardware applications (HyperLogLog, kmeans, AES, decision trees, sha256) and all microbenchmarks can be found under hw/hdl/operators. Examples of specific network operators are supplied as well. Default configurations of every operator coincide with ones used to obtain the results in the paper.

The code in sw/base is used for the tests where no explicit operator control is needed (AES, HyperLogLog, sha256). The code in sw/scheduling is used for the measurements of the reconfiguration time. Separate code for the operators requiring more control is provided (kmeans and decision trees).

A.6 Experiment customization

A wide variety of test cases and customization is available through different system configurations. The users can create different versions of the system through combinations of vFPGAs, network and memory stacks.

A.7 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts





Assise: Performance and Availability via Client-local NVM in a Distributed File System

Thomas E. Anderson[1] Marco Canini[2] Jongyul Kim[3†] Dejan Kostić[4] Youngjin Kwon[3]
Simon Peter[5] Waleed Reda[4,6⋆] Henry N. Schuh[1†] Emmett Witchel[5]

[1]*University of Washington* [2]*KAUST* [3]*KAIST* [4]*KTH Royal Institute of Technology*
[5]*The University of Texas at Austin* [6]*Université catholique de Louvain*

Abstract

The adoption of low latency persistent memory modules (PMMs) upends the long-established model of remote storage for distributed file systems. Instead, by colocating computation with PMM storage, we can provide applications with much higher IO performance, sub-second application failover, and strong consistency. To demonstrate this, we built the Assise distributed file system, based on a persistent, replicated coherence protocol that manages client-local PMM as a *linearizable* and *crash-recoverable* cache between applications and slower (and possibly remote) storage. Assise maximizes locality for all file IO by carrying out IO on process-local, socket-local, and client-local PMM whenever possible. Assise minimizes coherence overhead by maintaining consistency at IO operation granularity, rather than at fixed block sizes.

We compare Assise to Ceph/BlueStore, NFS, and Octopus on a cluster with Intel Optane DC PMMs and SSDs for common cloud applications and benchmarks, such as LevelDB, Postfix, and FileBench. We find that Assise improves write latency up to 22×, throughput up to 56×, fail-over time up to 103×, and scales up to 6× better than its counterparts, while providing stronger consistency semantics.

1 Introduction

Byte-addressable non-volatile memory (NVM), such as Intel's Optane DC persistent memory module (PMM) [14], is now commercially available as main memory. NVM provides high-capacity persistent memory with near-DRAM performance at lower cost. The promise of NVM as a low-cost main memory add-on is driving the adoption of node-local NVM at scale [43, 86, 87]. Remote direct memory access (RDMA) allows NVM access across the network without CPU overhead, raising interest in NVM for high-performance distributed storage.

A common paradigm in distributed file systems, like Amazon EFS [2], NFS [39], Ceph [82], Colossus/GFS [37], and NVM re-designs, like Octopus [58] and Orion [85], is to separate storage servers from clients. In this server-client design, files are stored by servers on machines physically separated from clients running applications. Client main memory is treated as a volatile block cache managed by the client's OS kernel. This design simplifies resource pooling by physically separating application from storage concerns with simple, server-managed data consistency mechanisms.

This simplicity comes at a cost, which becomes apparent as we move from SSD/HDD to NVM storage. In steady state, application performance is limited by the overhead to access kernel-level client caches. Upon a cache miss, multiple network round trips are needed to consult remote metadata servers and to fetch the actual data. On failure, client-server file systems must rebuild caches of failed clients from scratch, involving long fail-over times to re-establish application-level service and necessitating high network utilization during recovery. Third, managing client caches at fixed page-block granularity amplifies the small IO operations typical of many distributed applications and increases cache coherence overhead when IO is larger than the block size. These costs prevent NVM from reaching its performance potential and have led some within the storage community to advocate for a complete redesign of the file system API [54, 72, 73, 88].

We present Assise, a distributed file system designed to maximize the use of *client-local* NVM without requiring a new API for high performance. Assise unleashes the performance of NVM via pervasive and persistent caching in process-local, socket-local, and node-local NVM. Assise accelerates POSIX file IO by orders of magnitude by leveraging client-local NVM without kernel involvement, block amplification, or unnecessary coherence overheads. Assise provides near-instantaneous application fail-over onto a *hot replica* that mirrors an application's local file system cache in the replica's local NVM. Assise reduces node recovery time by orders of magnitude by locally recovering NVM caches with strong consistency semantics. Finally, Assise leverages cluster-wide NVM via *warm replicas* that provide lower latency reads than slower storage media, such as SSDs. In cascaded hot replica failure scenarios, warm replicas can become hot replicas to preserve near-instantaneous fail-over.

To enable these properties, we design and build to our knowledge the first crash consistent distributed file system cache coherence layer for replicated NVM (CC-NVM). CC-NVM serves cached file system state in Assise with strong consistency guarantees and locality. CC-NVM provides prefix crash consistency [80] by enforcing write order to local

⋆Lead student author.
†Co-student authors.

NVM via logging and to cross-socket and remote NVM by leveraging the write order of DMA and RDMA, respectively. CC-NVM provides linearizability for all IO operations via leases [38] that can be delegated among nodes, sockets, and processes for local management of file system state. CC-NVM consistently chain-replicates [77] all file system updates to a configurable set of hot and warm replicas for availability.

Using CC-NVM, Assise achieves the following goals:

- **Simple programming model.** Assise supports unmodified applications using the familiar POSIX API with strong linearizability and crash consistency [80].
- **Scalability.** Unlike NVM-aware distributed file systems that are limited to rack-scale [71, 85], Assise provides strong consistency but remains scalable using dynamic delegation of leases to nodes, sockets, and processes; local sharing uses CC-NVM for consistency without network, cross-socket, or kernel communication.
- **Low IO tail latency.** To efficiently support applications with low tail latency requirements, Assise allows kernel-bypass access to authorized local and remote NVM areas. To reduce write latency with replicated persistence, Assise provides an optimistic mode using asynchronous chain replication with prefix crash consistency.
- **High availability.** Assise provides near-instantaneous fail-over to a configurable number of replicas and minimizes the time to restore the replication factor after failure.
- **Efficient bandwidth use.** The high bandwidth provided by NVM means that communication can be a throughput bottleneck (cf. Table 1). Assise minimizes communication by eliminating redundant writes [52] and reducing coherence protocol overhead via logging.

We make the following contributions:

- We present the design (§3) and implementation (§4) of Assise, a distributed file system that fully utilizes NVM by persistent caching in client-local NVM as a primary design principle. Assise uses client-local NVM to recover the file system cache for fast fail-over and locally synchronizes reads and writes to file system state.
- We present CC-NVM (§3.3), the first persistent and available distributed cache coherence layer. CC-NVM provides locality for data and metadata access, replicates for availability, and provides linearizability and prefix crash consistency for all file system IO.
- We quantify the performance benefits of using local NVM versus remote NVM for distributed file systems (§5). We compare Assise's steady-state and fail-over behavior to RDMA-accelerated versions of Ceph with BlueStore [21] and NFS, as well as Octopus [58], a distributed file system designed for RDMA and NVM, using common cloud applications and benchmarks, such as LevelDB, Postfix, MinuteSort, and FileBench.

Our evaluation shows that Assise provides up to 22× lower write latency and up to 56× higher throughput than NFS and Ceph/BlueStore. Assise outperforms Octopus by up to an order of magnitude. Assise scales better than Ceph, providing 6× throughput for Postfix with 48 processes over 3 nodes. Assise is more available than Ceph, returning a recovering LevelDB store to full performance up to 103× faster. Demonstrating that strong consistency with the familiar POSIX API and high performance are not mutually exclusive, Assise finishes a local external sort 3% faster than a hand-tuned implementation using processor loads and stores to memory mapped NVM. Finally, Assise finishes the MinuteSort distributed sorting benchmark up to 2.2× faster than a parallel NFS installation.

Memory	R/W Latency	Seq. R/W GB/s	$/GB
DDR4 DRAM	82 ns	107 / 80	9.77 [19]
NVM (local)	175 / 94 ns	32 / 11.2	3.83 [20]
NVM-NUMA	230 ns	4.8 / 7.4	-
NVM-kernel	0.6 / 1 μs	-	-
NVM-RDMA	3 / 8 μs	3.8	-
SSD (local)	10 μs	2.4 / 2.0	0.32 [15]

Table 1: Memory & storage price/performance (October 2020).

Assise supports networked access to remote storage where it makes sense. Assise can automatically migrate cold data that does not fit in NVM to slower, network-attached storage devices, such as SSDs and HDDs. To do so, Assise's implementation builds on Strata [52] as its node-local store.

2 Background

Distributed applications have diverse workloads, with IO granularities large and small [56], different sharding patterns, and consistency requirements. All demand high availability and scalability. Supporting these properties simultaneously has been the focus of decades of distributed storage research [23, 39, 41, 58, 81, 82, 85]. Before NVM, trade-offs had to be made. For example, by favoring large transfers ahead of small IO, or steady-state performance ahead of crash consistency and fast recovery, leading to the common idiom of remote-storage file system design. We argue that with the arrival of fast NVM, these trade-offs need to be re-evaluated.

The opportunity posed by NVM is two-fold:

Cost/performance. Table 1 shows measured access latency, bandwidth, and cost for modern server memory and storage technologies, including Optane DC PMM (measurement details in §5). We can see that local NVM access latency and bandwidth are close to DRAM, up to two orders of magnitude better than SSD. At the same time, NVM's per-GB cost is only 39% that of DRAM. NVM's unique characteristics allow it to be used as the top layer in the storage hierarchy, as well as the bottom layer in a server's memory hierarchy.

Fast recovery. Persistent local storage with near-DRAM performance can provide a *recoverable* cache for hot file system data that can persist across reboots. The vast majority of system failures are due to software crashes that simply require rebooting [25, 36, 40]. Caching hot file system data in NVM allows for quick recovery from these common failures.

For these reasons, data center operators are deploying NVM

at scale [43, 86, 87]. However, to fully realize its potential, we have to efficiently use local NVM. NVM accessed via RDMA (NVM-RDMA), via loads and stores to another CPU socket (NVM-NUMA), or via the kernel on the same socket (NVM-kernel) can be an order of magnitude slower in terms of latency and bandwidth.

2.1 Related Work

We survey the existing work in distributed storage and highlight why it cannot fully utilize the storage system performance offered by local NVM.

Block and object stores, such as Amazon's EBS [1], S3 [3], and Ursa [56], provide a new API to a multi-layer storage hierarchy that can provide cheap, fault-tolerant access to vast amounts of data. However, block stores have a minimum IO granularity (16KB for EBS) and IO smaller than the block size suffers performance degradation from write amplification [56, 69]. For this reason, Dropbox uses Amazon S3 for data blocks, but keeps small metadata in DRAM for fast access, backed by an SSD [62]. Apache Crail [4] and Blizzard [63] provide file system APIs on top of block stores, but both focus on parallel throughput of large data streams, rather than small IO.

To realize the performance benefits of NVM for all IO, we need to abandon fixed block sizes and instead persist and track IO at its original operation granularity. Hence, Assise leverages logging to persist writes at their original granularity in NVM. A similar model is realized in the RAMcloud [66] key-value store. RAMcloud maintains data in DRAM for performance, using SSDs for asynchronous persistence. However, the capacity limits of DRAM mean that many RAMcloud operations still involve the network, and because DRAM state cannot be recovered after a crash, it is vulnerable to cascading node failures. Even after single node failures, state must be restored from remote nodes and RAMcloud requires a full-bisection bandwidth network for fast recovery. Assise leverages local NVM for recovery and does not require full-bisection bandwidth or asynchronous backup storage.

Client-server file systems, like Ceph [82], use distributed hashing over nodes to provide scalable file service for cloud applications. However, network and system call latency harms file IO latency, as shown in Table 1. Typical network access bandwidth to NVM is similarly surpassed by the higher bandwidth of local NVM.

To combat network overheads, several file systems have been built [58, 85] or retrofitted [13, 39, 44] to use RDMA. Octopus [58] and Orion [85] are redesigns that use RDMA for high performance access to NVM. Still, neither leverages kernel-bypass for low-latency IO (Octopus uses FUSE, Orion runs in the kernel) and both pool storage remotely. Like Ceph, Octopus uses distributed hashing to place files on nodes (Octopus does not replicate). Orion can store data locally via "internal clients," but uses a metadata server. Clover [76] is a key-value store that takes the opposite approach, locating metadata with applications, but storing data remotely. All systems perform remote operations in the common case to update data and/or metadata, increasing IO latency.

Concept	Explanation
LibFS	Per-process, user-level file system library
SharedFS	Per-socket system daemon; manages local leases
CC-NVM	Crash-consistent cache coherence with linearizability
Hot replica	Cache-hot replica for fast failover
Warm replica	Provides NVM for low-latency, remote, warm reads
Cluster manager	Fault-tolerant service for membership & leases

Table 2: Concepts used in Assise.

Network latency and limited bandwidth increase operation latency, reduce throughput, and limit scalability. For example, due to update contention at a central metadata server, Orion scales only to a small number of clients. Orion omits an evaluation of server fail-over and recovery (Assise's is in §5.4). Tachyon [55] aims to circumvent replication overhead by leveraging the concept of *lineage*, where lost output is recovered by re-executing application code that created the output. However, to do so, Tachyon requires applications to use a complex data lineage tracking API.

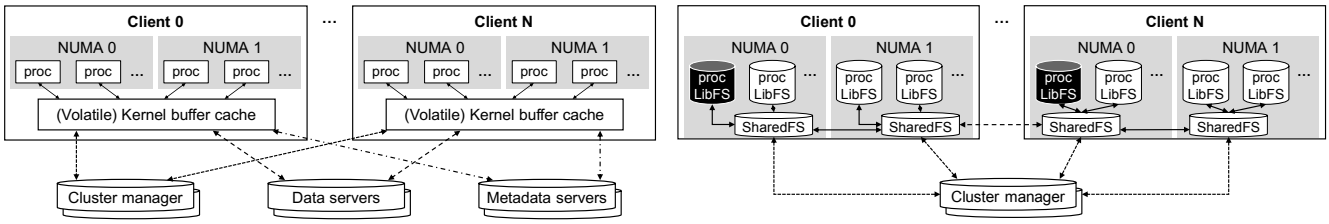
To maximize NVM utility, we need to design for a scenario where kernel and networking overheads are high compared to storage access. Hence, Assise eliminates kernel overhead for local IO operations and remote IO incurs a single operation to the nearest replica in the common case, without requiring dedicated metadata servers or a distributed hash to balance load. For scalability, we need to enforce data and metadata consistency locally, which CC-NVM tackles with the help of leases. Unlike Tachyon, Assise supports the classic POSIX file API and is fully compatible with existing applications.

Leases [38, 57] have long been integral to performance in distributed file systems, by allowing local operations to leased portions of the file system name space, with linearizability. Read-only leases are a common design pattern [12, 27, 39, 42], but some research systems have explored using both read and write leases in a similar manner to Assise. A prominent example is Berkeley xFS [23], which maintained a local block-level update log at each node, written as a software RAID 5/6 partitioned across other nodes. Assise differs from xFS by using an operational log, replicating rather than striping the log, and by doing update coalescing.

2.2 Remote Storage versus Local NVM

Figure 1 contrasts the IO architecture of traditional client-server file systems and Assise. Each subfigure shows two dual-socket nodes executing a number of application processes sharing a distributed file system. Both designs use a replicated cluster manager for membership management and failure detection, but they diverge in all other respects.

Traditional distributed file systems first partition available cluster nodes into clients and servers. Clients cache file system state in a volatile kernel buffer cache that is shared by processors across sockets (NVM-NUMA) and accessed via expensive system calls (NVM-kernel). Persistent file system state is stored in NVM on remote servers. For persistence and



(a) Traditional distributed FSes with server-side storage (NFS, Ceph, . . .).

(b) Client-local NVM (Assise).

Figure 1: Distributed file system IO architectures. Arrow = RPC/system call. Cylinder = persistence. Black = hot replica.

consistency, clients thus have to coordinate updates with replicated storage and metadata servers via the network (NVM-RDMA) with higher latency than local NVM. The cluster manager is not involved in IO. Data is typically distributed at random over replicated storage servers for simplicity and load balance [82]. The overhead of updating a large set of storage nodes atomically means that (crash) consistency is often provided only for metadata, which is centralized.

3 Assise Design

Assise avoids remote storage servers and instead uses CC-NVM to coordinate linearizable state among processes. Processes access cached file system state in local NVM directly via a library file system (LibFS), which may be replicated for fail-over (two LibFS hot replicas shown in black in Figure 1). CC-NVM coordinates LibFSes hierarchically via per-socket daemons (SharedFS) and the cluster manager. Table 2 explains several Assise-related concepts.

Crash consistency modes. Assise supports two crash consistency modes: optimistic or pessimistic [30]. Mount options specify the chosen crash consistency mode. When pessimistic, `fsync` forces immediate, synchronous replication and all writes prior to an `fsync` persist across failures. When optimistic, Assise commits all operations in order, but it is free to delay replication until the application forces it with a `dsync` call [30]. Optimistic mode provides lower latency persistence with higher throughput, but risks data loss after crashes that cannot recover locally (§3.4). In either mode, Assise guarantees a prefix crash-consistent file system [80]—all recoverable writes are in order and no parts of a prefix of the write history are missing.

We now describe cluster coordination and membership management in Assise (§3.1). We then detail the IO paths (§3.2) and show how CC-NVM interacts with them to provide linearizability and prefix crash consistency (§3.3). Finally, we describe recovery (§3.4) and warm replicas (§3.5). We close with a discussion of connected design questions (§3.6).

3.1 Cluster Coordination and Failure Detection

Like other distributed file systems, Assise leverages a replicated cluster manager for storing the cluster configuration and detecting node failures. Assise uses the ZooKeeper [10] distributed coordination service as its cluster manager.

Cluster coordination. Each SharedFS in Assise registers with the cluster manager. In our prototype, the system administrator decides which SharedFS replicates which parts of the cached file system namespace and the caching policy (hot or warm replica) for arbitrary subtrees; the cluster manager records this mapping. When a subtree is first accessed, LibFSes contact their local SharedFS, which consults the cluster configuration and sets up an RDMA replication chain from LibFS through the subtree's hot replicas. For each chain, hot replicas preallocate a configurable amount of NVM for replication (sensitivity evaluated in §5.2). It is future work to implement a distributed replica discovery service (e.g., using CC-NVM). LibFSes on any node are already able to cache any (meta-)data with linearizability.

Failure detection. The cluster manager sends heartbeat messages to each active SharedFS once every second. If no response is received after a timeout, the node is marked failed and all connected SharedFS are notified. When the node comes back online, it contacts the cluster manager and initiates recovery (§3.4).

3.2 IO Paths

Application IO interacts first with Assise caches. To keep tail latency low, Assise does not use a shared kernel buffer cache. Instead, LibFS caches file system state first in process-local memory. The LibFS cache uses both NVM and DRAM. NVM stores updates, while DRAM caches reads. LibFS implements the POSIX API at user-level. We now discuss cache operation upon IO, including replication, eviction, and access permissions. Figure 2 shows these mechanisms for two hot replicas and one warm replica, using SSDs for cold storage. Cache coherence is discussed in §3.3.

3.2.1 Write Path

Writes in Assise involve three mechanisms that operate on different time scales:

1. To allow for persistence with low latency, LibFS directly writes into a process-local cache in NVM (Ⓦ). To efficiently support writes of any granularity, the write cache is an *update log*, rather than a block cache.
2. To outlive node failures, the update log is chain-replicated, with kernel-bypass, by LibFS (Ⓢ1, Ⓢ2).
3. When update logs fill beyond a threshold, evictions are initiated (Ⓔ2), moving their contents to SharedFS. We describe replication and eviction next.

Replication and crash consistency. When pessimistic, `fsync` forces immediate, synchronous replication. The caller

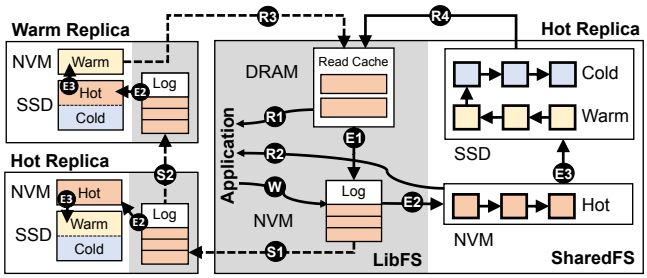


Figure 2: Assise IO paths. Dashed line = RDMA operation, solid line = local operation. Shaded areas are per process.

is blocked until all writes up to the `fsync` have been replicated. Thus, all writes prior to an `fsync` outlive node failures.

When optimistic, Assise is free to delay replication. This provides Assise with an opportunity to *coalesce* [52] temporary durable writes (i.e., overwritten or deleted files), a workload pattern seen in application-level commit protocols [67]. Eliminating these writes allows Assise to conserve network bandwidth. In optimistic mode, Assise initiates replication on `dsync` or upon log eviction.

In both cases, the local update log contents are written to preallocated NVM on the first replica along the replication chain via RDMA (S1). The replica continues chain replication to the next replica (S2), and so on. The final replica in the chain sends an acknowledgment back along the chain to indicate that the chain completed successfully.

Cache eviction. When a LibFS update log fills, it replicates any unreplicated writes and initiates eviction. Eviction is done in least-recently-used (LRU) fashion through the SharedFS shared caches to cold storage (E2, E3). Hot replicas keep *hot* data in NVM, while moving *warm* and *cold* data to cold storage. Warm replicas (§3.5) keep hot and cold data in cold storage, while warm data resides in NVM to accelerate warm reads (§3.5). Cold storage may be remote (e.g., via NVMe-over-Fabrics [17]). Each replica along the chain evicts in parallel and acknowledges when eviction is finished. This ensures that all replicas cache identical state for fast failover.

For log eviction (E2), issuing direct stores to NVM shared caches on another socket has overhead due to cross-socket hardware cache coherence, limiting throughput [83]. Since CC-NVM provides cache coherence, Assise can bypass hardware cache coherence by using DMA [53] when evicting to NVM-NUMA. This yields up to 30% improvement in cross-socket file system write throughput (§5.5).

3.2.2 Read Path

LRU cache eviction guarantees that the latest version of all data is always available in the fastest cache. Thus, upon a read, LibFS (1) checks the process-private write and read caches (via a *log hashtable* and *read cache*, shown in Figure 2) for the requested data (R1). If not found, LibFS (2) checks the node-local hot SharedFS cache (R2) (via an *extent tree* used to index the SharedFS cache [52]). If the data was found in either of these areas, it is read locally. If not found, LibFS (3) checks the warm replica's SharedFS cache (R3), if it exists, and, in parallel, checks cold storage (R4).

Read cache management. Recently read data is cached in process-local DRAM, except if it was read from local NVM, where DRAM caching does not provide benefit. LibFS prefetches up to 256KB from cold storage and up to 4KB from remote NVM. For remote NVM reads, LibFS first fetches the requested data and then prefetches the remainder. This minimizes small read latency while improving the performance of workloads with spatial locality. Data from remote NVM and cold storage is evicted from the read cache to the process-local update log (E1).

3.2.3 Permissions and Kernel Bypass

Assise assumes a single administrative domain with UNIX file and directory ownership and permissions. SharedFS enforces that LibFS may access only authorized data, by checking permissions and metadata integrity upon cache eviction and enforcing permissions on reads. To minimize latency of node-local SharedFS cache reads, Assise allows read-only mapping of authorized parts of the SharedFS cache into the LibFS address space. LibFS caches and mappings are invalidated when files or directories are closed and whenever the update log is evicted.

The metadata integrity of the file system is ensured by SharedFS. LibFS operations do not prevent one thread from corrupting another's data in the process-local update log, but SharedFS verifies that all metadata operations are valid before they become visible to other processes. This implies that processes can corrupt *their* own data in their private update log, even after it was written (memory protection keys can mitigate inter-thread data corruption [34]). However, only process-local writes go to the process-local update logs. Multi-process access to any filesystem object (including a subtree) is linearizable and access-controlled via leases. Processes cannot corrupt shared file system (meta-)data.

3.3 Crash Consistent Cache Coherence with CC-NVM

CC-NVM provides distributed cache coherence with linearizability when sharing file system state among processes; it provides prefix semantics upon a crash.

Prefix crash consistency. To provide prefix crash consistency, CC-NVM tracks write order via the update log in process-local NVM. Each POSIX call that updates state is recorded, in order, in the update log. When chain-replicating, CC-NVM leverages the write ordering guarantees of (R)DMA to write the log in order to replicas. In optimistic mode, CC-NVM wraps coalesced file system operations in a Strata transaction [52]. This ensures that file system updates are persisted and replicated atomically and that a prefix of the write history can be recovered (§3.4).

Sharing with linearizability. CC-NVM serializes concurrent access to shared state by multiple processes and recovers the same serialization after a crash via leases [38]. Leases

provide a simple, fault-tolerant mechanism to delegate access. Leases function similarly to reader-writer locks, but can be revoked (to allow another process access) and expire after a timeout (after which they may be reacquired). In CC-NVM, leases are used to grant shared read or exclusive write access to a set of files and directories—multiple read leases to the same set may be concurrently valid, but write leases are exclusive. Reader/writer semantics efficiently support shared files and directories that are read-mostly and widely used, but also write-intensive files and directories that are not frequently shared. CC-NVM also supports a *subtree lease* that includes all files and directories at or below a particular directory. A subtree lease holder controls access to files and directories within that subtree. For example, a LibFS with an exclusive subtree lease on `/tmp/bwl-ssh/` can recursively create and modify files and directories within this subtree.

Leases must be acquired by LibFS from SharedFS via a system call before LibFS can cache the data covered by the lease. Assise does this upon first IO; leases are kept until they are revoked by SharedFS. This occurs when another LibFS wishes access to a leased file or when a LibFS instance crashes or the lease times out. Lease revocation latency is bounded by a grace period, within which the current lease holder can finish its ongoing IO before releasing contended leases. If LibFS fails to surrender the lease after the grace period, the lease is revoked by SharedFS and any subsequent IO on the leased file is rejected as invalid. SharedFS enforces that the lease holder's read and write caches are cleaned and evicted of the covered data before the lease is transferred. The time taken to do so is bounded by the holder's update log size. SharedFS logs and replicates each lease transfer in NVM for crash consistency. A LibFS may overlap IO with SharedFS lease replication until `fsync`/`dsync`.

Hierarchical coherence. To localize coherence enforcement, leases are delegated hierarchically. The cluster manager is at the root of the delegation tree, with SharedFSes as children, and LibFSes as leaves (cf. Figure 1b). LibFSes request leases first from their local SharedFS. If the local SharedFS is not the lease holder, it consults the cluster manager. If there is no current lease holder, the cluster manager assigns the lease to the requesting SharedFS, which delegates it to the requesting LibFS and becomes its *lease manager*. If a lease manager already exists, SharedFS forwards the request to the manager and caches the lease manager's information (leased namespace and expiration time of lease). The cluster manager expires lease management from SharedFSes every 5 seconds. This allows CC-NVM to migrate lease management to the local SharedFS, while preventing leases from changing managers too quickly, facilitating scalability.

Hierarchical coherence minimizes network communication and thus lease delegation overhead. LibFSes on the same node or socket require only local SharedFS delegation in the common case. This structure maps well to the data sharding patterns of many distributed applications (§5.5).

3.4 Fail-over and Recovery

Assise caches file system state with persistence in local NVM, which it can use for fast recovery. Assise optimizes recovery performance for the most common crash types.

LibFS recovery. An application process crashing is the most common failure scenario. In this case, the local SharedFS simply evicts the dead LibFS update log, recovering all completed writes (even in optimistic mode) and then expires its leases. Log-based eviction is idempotent [52], ensuring consistency in the face of a system crash during eviction. The crashed process can be restarted on the local node and immediately reuse all file system state. The LibFS DRAM read-only cache has to be rebuilt, with minimal performance impact (§5.4).

SharedFS recovery. Another common failure mode is a reboot due to an OS crash. In this case, we can use NVM to dramatically accelerate OS reboot by storing a checkpoint of a freshly booted OS. After boot, Assise can initiate recovery for all previously running LibFS instances, by examining the SharedFS log stored in NVM.

Cache replica fail-over. To avoid waiting for node recovery after a power failure or hardware problem, we immediately fail-over to a hot replica. The replica's SharedFS takes over lease management from the failed node, using the replicated SharedFS log to re-grant leases to any application replicas. The new instances will see all IO that preceded the most recently completed `fsync`/`dsync`.

Writes to the file system can invalidate cached data of the failed node during its downtime. To track writes, the cluster manager maintains an epoch number, which it increments on node failure and recovery. All SharedFS instances are notified of epoch updates. All SharedFS instances share a per-epoch bitmap in a sparse file indicating what inodes have been written during each epoch. The bitmaps are deleted at the end of an epoch when all nodes have recovered.

Node recovery. When a node crashes, the cluster manager makes sure that all of the node's leases expire before the node can rejoin. When rejoining, Assise initiates SharedFS recovery. A recovering SharedFS contacts an online SharedFS to collect relevant epoch bitmaps. SharedFS then invalidates every block from every file that has been written since its crash. This simple protocol could be optimized, for instance, by tracking what blocks were written, or checksumming regions of the file to allow a recovering SharedFS to preserve more of its local data. But the table of files written during an epoch is small and quickly updated during file system operation, and our simple policy has been sufficient.

3.5 Warm Replicas

To fully exploit the memory hierarchy presented in Table 1, remote NVM can be used as a third-level cache, below local DRAM and local NVM. To do so, we introduce *warm replicas*. Like hot replicas, warm replicas receive all file system updates via chain-replication, but leverage a different update

log eviction policy. Warm replicas track the LRU chain for a specified portion of "warm data" beyond the LibFS and SharedFS caches. Warm replicas do not impact the latency of replicated writes, but they reduce read latency for warm data by serving these reads from NVM, rather than cold storage.

LibFSes can read from warm replicas via RDMA with lower latency and higher bandwidth than cold storage (NVM-RDMA versus SSD in Table 1). Applications do not run on warm replicas in the common case. In the rare case of a failure cascade crashing all hot replicas, processes can fail-over to warm replicas, albeit with reduced short-term performance. After fail-over, warm replicas become hot replicas and hot data must be migrated back into local NVM.

3.6 Discussion

Assise may be deployed at scale. The use of local NVM together with hierarchical lease delegation aligns well with datacenter server, rack, and pod architecture [22]. We discuss factors of Assise's design that impact such a deployment. In particular, the memory overhead of per-process and per-replica update logs, the use of NVM and RDMA at scale, and security.

Update log scalability. Assise uses per-process and per-replica update logs for efficient chain-replication with kernel-bypass. These update logs are preallocated on process creation in our prototype. While update logs can support high performance at moderate size (§5.2), a deployment at scale might be concerned with the memory consumption of update logs. In this case, the per-process and per-replica update log size can be adapted dynamically to momentarily available NVM capacity and per-process IO demand. SharedFS can resize logs upon eviction. The most significant overhead for log resizing is memory registration for RDMA. It requires pinning the memory and mapping it in the RDMA NICs. This operation can be overlapped with the log eviction itself. To help reduce the need for frequent resizing, logs can be resized multiplicatively, similar to resizing approaches in prior work [84].

RDMA scalability. Assise uses RDMA reliable connections (RCs) for each process and replica. RCs require the NIC to create and maintain connection state. For larger clusters, maintaining a large number of connections can stress the NIC's limited memory and degrade performance. Several proposals have been made to reduce NIC cache thrashing [29, 68] and Mellanox introduced dynamically-connected (DC) transports [70], which allows connection-sharing and enables a high degree of scalability. Assise can leverage these approaches to scale the use of RDMA.

NVM wear-out. Assise uses local NVM extensively. This use can lead to the wear-out of NVM. To prevent frequent NVM replacement at scale, it is important to minimize writes to the NVM media. Assise's update logs minimize write amplification, but update log eviction in causes a 2× write amplification in the worst case. This write amplification can be partially eliminated via coalescing as seen in workloads, like Varmail (§5.3). To further reduce write amplification, update log pages may be remapped to the SharedFS shared cache, without introducing any additional writes [48]. We leave this as future work.

Security. In a large-scale public cloud scenario, data from each tenant is usually encrypted for security. For this purpose, both NVM and RDMA support encryption of data at rest and in-flight. Intel's Optane DC PMMs support transparent hardware encryption of data stored in NVM and modern RDMA NICs [61] support transparent encryption of RDMA operations.

4 Implementation

Assise uses *libpmem* [11] for persisting data on NVM and *libibverbs* for RDMA operations in userspace. Assise intercepts POSIX file system calls and invokes the corresponding LibFS implementation of these functions in userspace [8]. The Assise implementation consists of 28,982 lines of C code (LoC), with LibFS and SharedFS using 16,515 and 6,563 LoC, respectively. The remaining 5,904 LoC contain utility code, such as hash tables and linked lists. SharedFS communicates with LibFSes via shared memory [24]. Assise uses Strata code (LoC not counted) for cold storage in SSD and HDD.

Assise uses Intel Optane DC PMM in App-Direct mode. App-Direct exposes NVM as a range of physical memory. It is the most efficient way to access NVM, but it requires OS support. OS-transparent modes have weaker persistence or performance properties [45]. For example, memory mode integrates NVM as *volatile* memory, using DRAM as a hardware-managed level 4 cache. Sector mode exposes NVM as a disk, with attendant IO amplification and disk driver overheads.

4.1 Strata as a Building Block

Assise builds upon Strata's local file system functionality and augments it with the CC-NVM cache coherence layer and RDMA to create a replicated and highly efficient distributed file system with prefix crash conistency. Assise inherits several components from Strata, including its use of extent trees to index storage managed by SharedFS (in turn based on Ext4 [60]), the LibFS update log, and log coalescing. We enhance Strata's extent trees to manage directories and Strata's leases to support delegation.

4.2 Efficient Network IO with RDMA

Assise makes efficient use of RDMA. For lossless, in-order data transfer among nodes, Assise uses RDMA reliable connections (RCs). RCs have low header overhead, improving throughput for small IO [49, 59]. RCs also provide access to one-sided verbs that bypass CPUs on the receiver side, reducing message transfer times [35, 64] and memory copies [74].

Log replication. Logs are naturally suited for one-sided RDMA operations. Replication typically requires only one RDMA write, reducing header and DMA overheads [59]. As-

sise uses RDMA *write-with-immediate* for log replication. This operation performs a write and also notifies the remote replica to forward the data to the next replica in the chain. The only exceptions are when the remote log wraps around or when the local log is fragmented (due to coalescing), such that it exceeds the NIC's limit for scatter-gather DMA.

Persistent RDMA writes. The RDMA specification does not define the persistence properties of remote NVM writes via RDMA. In practice, the remote CPU is required to flush any RDMA writes from its cache to NVM. Assise flushes all writes via the CLWB and SFENCE instructions on each replica, before acknowledging successful replication. In the future, it is likely that enhancements to PCIe will allow RDMA NICs to bypass the processor cache and write directly to NVM to provide persistence without CPU support [50].

Remote NVM reads. Assise reads remote data via RPC. To keep the request sizes small, Assise identifies files using their inode numbers instead of their path. As an optimization, DRAM read cache locations are pre-registered with the NIC. This allows the remote node to reply to a read RPC by RDMA writing the data directly to the requester's cache, obviating the need for an additional data copy.

5 Evaluation

We evaluate Assise's common-case as well as its recovery performance, and break down the performance benefits attained by each system component. We compare Assise to three state-of-the-art distributed file systems that support NVM and RDMA. Our experiments rely on several microbenchmarks and Filebench [75] profiles, in addition to several real applications, such as LevelDB, Postfix, and MinuteSort. Our evaluation answers the following questions:

- **IO latency and throughput breakdown (§5.2).** What is the hardware IO performance of a storage hierarchy with local NVM (Table 1)? How close to this performance do the file systems operate under various IO patterns? What are the sources of overhead?
- **Cloud application performance (§5.3).** What is the performance of cloud applications with various consistency, latency, throughput, and scalability requirements? What is the overhead of Assise's POSIX API implementation versus hand-tuned, direct use of local NVM? By how much can a warm replica improve read latency? By how much can optimistic crash consistency improve write throughput for real applications?
- **Availability (§5.4).** How quickly can applications recover from various failure scenarios?
- **Scalability (§5.5).** How well does Assise perform when multiple processes share the file system? By how much can Assise's hierarchical coherence improve multi-process, multi-socket, and multi-node scalability?

Testbed. Our experimental testbed consists of 5× dual-socket Intel Cascade Lake-SP servers running at 2.2GHz, with a total of 48 cores (96 hyperthreads), 384GB DDR4-2666 DRAM, 6TB Intel Optane DC PMM, 375GB Intel Optane DC P4800X series NVMe-SSD, and a 40GbE ConnectX-3 Mellanox InfiniBand NIC, connected via an InfiniBand switch. Exploiting all 6 memory channels per processor, there are 6 DIMMs of DRAM and NVM per socket. NVM is used in App-Direct mode (§4). All nodes run Fedora 27 with Linux kernel version 4.18.19.

Feature	Assise	Ceph	NFS	Octopus	Orion
Cache recovery	✓				
Local consistency	✓				
Kernel-bypass	✓				
Linearizability	✓				✓
Data crash consistency	✓				✓
Byte-oriented	✓			✓	✓
Replication	✓	✓			✓
RDMA	✓	✓	✓	✓	✓

Table 3: Features of the evaluated distributed file systems.

Hardware performance. We first measure the achievable IO latency and throughput for each memory layer in our testbed server. We do this by using sequential IO and as many cores of a single socket as necessary. We measure DRAM and NVM (App-Direct) latency and throughput using Intel's memory latency checker [5]. NVM-RDMA performance is measured using RDMA read and write-with-immediate (to flush remote processor caches) operations to remote NVM. SSD performance is measured using `/dev/nvme` device files. The IO sizes that yielded maximum performance are 64B for DRAM, 256B for NVM, and 4KB for SSD. Table 1 shows these results. The measured IO performance for DRAM, NVM, and SSD matches the hardware specifications of the corresponding devices and is confirmed by others [45]. NVM-RDMA throughput matches the line rate of the NIC. NVM-RDMA write latency has to invoke the remote CPU (to flush caches) and is thus larger than read latency. We now investigate how close to these limits each file system can operate.

State-of-the-art. Table 3 shows performance-relevant features of the state-of-the-art and Assise. We can see that no open-source distributed file system provides all of Assise's features. Hence, a direct performance comparison is difficult. We perform comparisons against the Linux kernel-provided NFS version 4 [39] and Ceph version 14.2.1 [82] with BlueStore [21], both retrofitted for RDMA, as well as Octopus [58]. We cannot directly compare with Orion [85] as it is not publicly available, but we emulate its behavior where possible. Only Ceph provides availability via replicated object storage daemons (OSDs), delegating metadata management to a (potentially sharded) metadata server (MDS). Octopus and NFS do not support replication for availability and thus gain an unfair performance advantage over Assise. However, Assise beats them even while replicating for availability, showing that both features can be had when leveraging local NVM.

Other file systems do not support persistent caches and their consistency semantics are often weaker than Assise's. Assise provides data crash consistency, while both Ceph/BlueStore

and Octopus provide only metadata crash consistency [31]. For NFS, crash consistency is determined by the underlying file system. We use EXT4-DAX [9], which also provides only metadata crash consistency. When sharing data, NFS provides *close-to-open consistency* [39], while Octopus and Ceph provide "stronger consistency than NFS" [28], and Assise provides linearizability, which is stronger than Octopus' and Ceph's guarantees. In all performance comparisons, Assise provides stronger consistency than the alternatives. Ceph is the closest comparison point.

File system compliance tests. We tested Assise using xfstests [18] and CrashMonkey [65]. Assise passed all 75 generic xfstests that are recommended for NFS [16]. NFSv4.2 and Ceph v14.2.1 pass only 71 and 69 of these tests, respectively. In part, this is due to their weaker consistency model. Assise also successfully passes CrashMonkey tests, runs all existing Filebench profiles, passes all unit tests for the LevelDB key-value store, and passes MinuteSort validation.

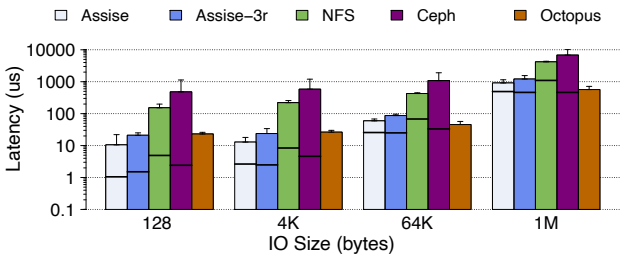
5.1 Experimental Configuration

Machines. Each experiment specifies the number (≥ 2) of testbed machines used. By default, machines are used as hot replicas in Assise, as a pool of storage nodes in Octopus, and as OSD and MDS replicas in Ceph. NFS uses only one machine as server, the rest as clients. We place applications on hot replicas for Assise, on OSD replicas for Ceph, on storage nodes for Octopus, and on clients for NFS. Assise's and Ceph's cluster managers run on 2 additional testbed machines (NFS and Octopus do not have cluster managers). The colocated deployment of applications and OSDs for Ceph is due to the small size of our cluster. It gives Ceph a potential performance advantage over an all-remote OSD deployment.

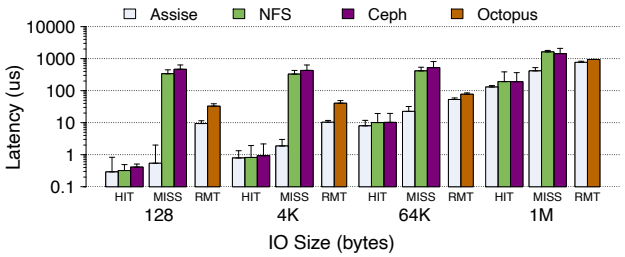
Network. We use RDMA for the NFS client-server connection. Ceph provides its client-side file system via the Ceph kernel driver and uses IP over InfiniBand, which was the fastest configuration (we also tried FUSE and Accelio [13]). Assise and Octopus use RDMA with kernel-bypass.

Storage and caches. For maximum efficiency, all file systems use NVM in App-Direct mode to provide persistence (cylinders in Figure 1) and DRAM when persistence is not needed (e.g., kernel buffer cache). We investigate Ceph and NFS performance using NVM in memory mode for volatile caches and find it to degrade throughput by up to 25% versus DRAM. For efficient access to NVM, Ceph OSDs use BlueStore and NFS servers use EXT4-DAX. Octopus uses FUSE to provide its file system interface to applications in *direct IO* mode to NVM, bypassing the kernel buffer cache [6].

To evaluate a breadth of cache behaviors with limited application data set sizes, we limit the fastest cache size for all file systems to 3GB. For Ceph and NFS, we limit the kernel buffer cache to 3GB. For Assise, we partition the LibFS cache into a 1GB NVM update log and a 2GB DRAM read cache (the SharedFS second-level cache may use all NVM available), and we run Assise in pessimistic mode.



(a) Sequential write. `write` latency is solid line, `fsync` is bar height.



(b) Read latencies for cache hits, misses, and remote (RMT) misses.

Figure 3: Avg. and 99%ile (error bar) IO latencies. Log scale.

5.2 Microbenchmarks

Average and tail write latency. We compare unladen synchronous write latencies with 2 machines (except Assise-3r which uses 3 machines). Synchronous writes involve `write` calls (fixed-width font identifies POSIX calls) that operate locally (except for Octopus where `write` may be remote), and `fsync` calls that involve remote nodes for replication (Assise, Ceph) and/or persistence (Ceph, NFS). Each experiment appends 1GB of data into a single file, and we report per-operation latency. In this case, the file size is smaller than each file system's cache size, so no evictions occur—with gigabytes of cache capacity, this is common for latency-sensitive write bursts.

Figure 3a shows the average and 99th percentile sequential write latencies over various common IO sizes (random write latencies are similar for all file systems). We break writes down into `write` (solid line) and `fsync` call latencies (bar). Octopus' `fsync` is a no-op. Assise's local write latencies match that of Strata [52]. Assise's average write latency for 128B two-node replicated writes is only 8% higher than the aggregate latencies of the required local and NVM-RDMA writes (cf. Table 1). Three replicas (Assise-3r) increase Assise's overhead to 2.2× due to chain-replication with sequential RPCs. The 99th percentile replicated write latency is up to 2.1× higher than the average for 2 replicas. This is due to Optane PMM write tail-latencies [45]. The tail difference diminishes to 19% for 3 replicas due to the higher average.

Ceph and NFS use the kernel buffer cache and interact at 4KB block granularity with servers. For small writes, the incurred network IO amplification during `fsync` is the main reason for up to an order of magnitude higher aggregate write latency than Assise. In this case, their `write` latency is up

to 3.2× higher than Assise due to kernel crossings and copy overheads. For large writes (≥ 64KB), network IO amplification diminishes but the memory copy required to maintain the kernel buffer cache becomes a major overhead. The latency of large `writes` is higher than Assise's replicated write latency (and up to 2.7× higher than Assise's non-replicated `write` latency), while aggregate write latency is up to 7.2× higher than Assise. Ceph has higher `fsync` latency than NFS due to replication.

Octopus eliminates the kernel buffer cache and block orientation, which improves its performance drastically versus NFS and Ceph. However, Octopus still treats all NVM as remote and uses FUSE for file IO. Octopus exhibits up to 2.1× higher latency than Assise for small (< 64KB) writes. This overhead stems from FUSE (around 10μs [78]) and writing to remote NVM via the network. Large writes (≥ 64KB) amortize Octopus' write overheads. Assise has up to 1.7× higher write latency due to replication. Octopus does not replicate.

Average and tail read latency. We compare unladen read latencies across different cache configurations. To do this, we read a 1GB file using various IO sizes, once with a warm cache (to report cache hits) and once with a cold cache (to report misses). The results are shown in Figure 3b. Assise has a second-layer cache in SharedFS before going remote, and we report three cases for Assise. Reads in Octopus are always remote.

We first compare cache-hit latencies (HIT), where Assise is up to 40% faster than NFS and 50% faster than Ceph. Assise serves data from the LibFS read cache, while NFS and Ceph use the kernel buffer cache. If Assise misses in the LibFS cache, data may be served from the local SharedFS (MISS). Assise-MISS incurs up to 3.2× higher latency than Assise-HIT due to reading the extent tree index, especially for larger IO sizes that read a greater number of extents. If Assise misses in both caches, it has to read from a remote replica (RMT). Assise-RMT incurs the latency of an RPC using RDMA. When NFS and Ceph miss in the cache, their clients have to fetch from remote servers, which incurs up to orders of magnitude higher average and tail latencies than Assise-RMT and Assise-MISS. Ceph performs worse than NFS due to a more complex OSD read path.

The elimination of a cache hurts Octopus' read performance, because it has to fetch metadata and data (serially) from remote NVM (RMT). Octopus' read latency is up to two orders of magnitude higher than the other file systems hitting in the cache, and up to an order of magnitude lower than NFS and Ceph missing in the cache. Octopus does not handle small (≤ 4KB) reads well due to FUSE overheads, with up to 3.54× Assise-RMT read latency. This overhead is amortized for larger reads (≥ 64KB), where Octopus incurs up to 1.46× the read latency of Assise-RMT. By configuring FUSE to use the kernel buffer cache for Octopus, we reduce Octopus' read hit latency to 1.8× that of Assise-HIT, with the remaining overhead due to FUSE. However, using the kernel buffer cache inflates write latencies for Octopus by up to an order of magnitude due to additional buffer cache memory copies.

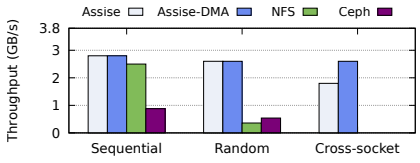
Peak throughput. Figure 4 shows average throughput of sequential and random IO to a 120GB dataset (on the local socket) with 4KB IO size from 24 threads (all cores of one socket). To evaluate a standard replication factor of 3, we use 3 machines for Assise and Ceph. The dataset is sharded over 24 files, and 5GB of data is written per thread. For random writes, a random offset is generated for every IO. `write` calls are not followed by `fsync` and the total amount of accessed data is larger than the cache size, causing cache eviction on write. The cache is initially cold so reads miss in the cache. For Assise, we show cache miss performance from a local and remote SharedFS. Octopus crashes during this experiment and is not shown.

For sequential writes, Assise and NFS achieve 74% and 66% of the NVM-RDMA bandwidth (cf. Table 1), respectively, due to protocol overhead for NFS and log header overhead for Assise. Chain-replication in Assise affects throughput only marginally. Ceph replicates in parallel to 2 remote replicas, consuming 3× the network bandwidth. This reduces its throughput to 31% of Assise and 35% of NFS. Assise achieves similar performance for sequential and random writes, as Assise's writes are log-structured. NFS and Ceph perform poorly for random writes due to cache block misprefetching incurring additional reads from remote servers, causing Assise to achieve 4.8× Ceph's throughput. NFS throughput is at only 67% that of Ceph, which is due to kernel locking overhead.

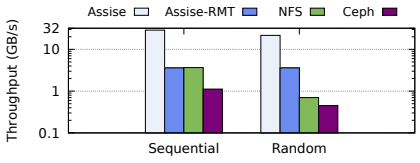
To quantify the benefit of bypassing hardware cache coherence for cross-socket writes with DMA, we repeat the benchmark, placing all files on the remote socket. We can see that Assise-DMA attains 44% higher cross-socket throughput than non-temporal processor writes (Assise). Sequential and random writes provide comparable performance. NVM-NUMA writes occur during eviction from the LibFS update log (local socket) to the NVM shared cache (remote socket). When writing to the local socket, Assise-DMA attains identical throughput to Assise, regardless of pattern.

For local sequential and random reads from the local SharedFS cache, Assise achieves 90% and 68%, respectively, of local, sequential NVM bandwidth. The 10% difference for sequential reads to local NVM bandwidth is due to metadata lookups, while random reads additionally suffer PMM buffer misses [45]. Assise remote reads (Assise-RMT) attain full NVM-RDMA bandwidth (3.8GB/s), regardless of access pattern. NFS and Ceph are limited by NVM-RDMA bandwidth for all reads and again have worse random read performance due to prefetching.

Log size sensitivity. To evaluate the impact of log size on write throughput, we conduct a sensitivity analysis. We run a single-process microbenchmark that writes a 1GB file sequentially at 4KB IO granularity. This experiment models a



(a) Write. 3.8GB/s is NVM-RDMA bandwidth.



(b) Read. 32GB/s is NVM read bandwidth.

Figure 4: Average throughput with 24 threads at 4KB IO size.

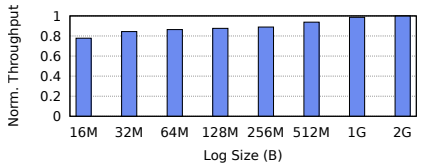


Figure 5: Worst-case throughput versus update log size, normalized to 2GB.

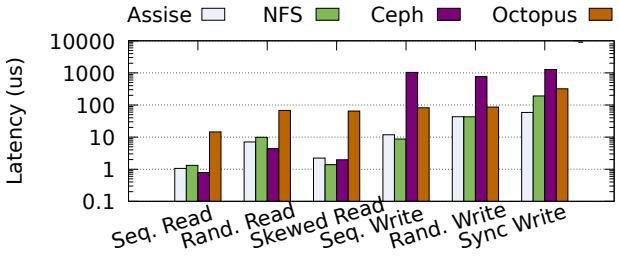


Figure 6: Average LevelDB benchmark latencies. Log scale.

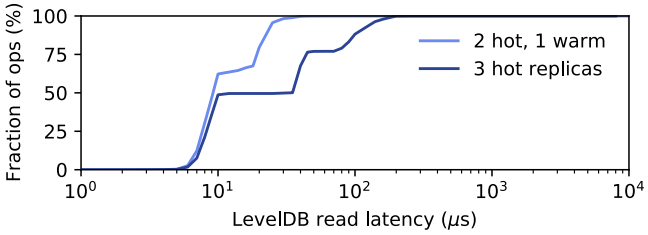


Figure 7: LevelDB random read latencies with warm replica.

worst case scenario. In the absence of sharing, processes can quickly fill up their allocated log space. Figure 5 shows the normalized write throughput at different log sizes. Throughput increases with log size, but the performance impact is small. Throughput increases by only 22% when using a 2GB log size versus a 16MB log size, a 128× increase in log size. For workloads that share data, we expect this gap to be smaller, as logs are evicted upon lease handoff. With 6TB of NVM per machine, Assise can scale to thousands of processes even with 2GB update logs. At 16MB, 100,000s of processes can be supported.

5.3 Application Benchmarks

We evaluate the performance of a number of common cloud applications, such as the LevelDB key-value store [33], the Fileserver and Varmail profiles of the Filebench [75] benchmarking suite, emulating file and mail servers, and the MinuteSort benchmark. We use 3 machines for LevelDB and Filebench and 5 machines for MinuteSort.

LevelDB. We run a number of single-threaded LevelDB latency benchmarks using LevelDB's `db_bench`, including sequential and random IO, skewed random reads with 1% of highly accessed keys, and sequential synchronous writes (`fsync` after each write). All benchmarks use a key size of 16B and a value size of 1KB with a working set of 1M KV pairs. Figure 6 presents the average measured operation latency, as reported by the benchmark.

Assise, Ceph, and NFS perform similarly for reads, where caching allows them to operate close to hardware speeds. For non-synchronous writes, NFS is up to 26% faster than Assise, as these go to its client kernel buffer cache in large batches (LevelDB has its own write buffer), while Assise is 69% faster than NFS for synchronous writes that cannot be buffered. Random IO and synchronous writes incur increasing LevelDB indexing overhead for all systems. Ceph performs worse than NFS for writes because it replicates (as does Assise) and Assise performs 22× better. Octopus bypasses the cache and thus performs worst for reads and better only than Ceph for writes, as it does not replicate.

Warm replica read latency. Warm replicas reduce read latency for warm data by allowing these reads to be served from remote NVM, rather than cold storage. For this benchmark, we configure Assise to limit the aggregate (LibFS and SharedFS) cache to 2GB and use the local SSD for cold storage. We then run the LevelDB random read experiment with a 3GB dataset. We repeat the experiment with two setups: (1) with 3 hot replicas and (2) with 2 hot and 1 warm replica. Figure 7 shows a CDF of read latencies. The benchmark accesses data uniformly at random, causing 33% of the reads to be warm. Consequently, at the 50th percentile, read latencies are similar for both configurations (served from cache). At the 66th percentile, reads in the first setup are served from SSD and have 2.2× higher latency than warm replica reads in the second setup. At the 90th percentile, the latency gap extends to 6×.

Filebench. Varmail and Fileserver operate on a working set of 10,000 files of 16KB and 128KB average size, respectively. Files grow via 16KB appends in both benchmarks (mail delivery in Varmail). Varmail reads entire files (mailbox reads) and Fileserver copies files. Varmail and Fileserver have write to read ratios of 1:1 and 2:1, respectively. Varmail leverages a write-ahead log with strict persistence semantics (`fsync` after log and mailbox writes), while Filebench consistency is relaxed (no `fsync`). Figure 8 shows average measured throughput of both benchmarks. Assise outperforms Octopus (the best alternative) by 6.7× for Fileserver and 5.1× for Varmail. Ceph performs worse than NFS for Varmail due to stricter persistence requiring it to replicate frequently and due to MDS contention, as Varmail is metadata intensive.

Optimistic crash consistency. We repeat this benchmark for Assise in optimistic mode (Assise-Opt) and change Varmail to use synchronous writes for the mailbox, but non-synchronous writes for the log. Prefix semantics allow Assise to buffer and

System	Processes	Partition [s]	Sort [s]	**Total [s]**	**GB/s**
Assise	160	20.3	43.0	**63.3**	**5.1**
	320	52.1	43.0	**95.1**	**6.7**
NFS	160	60.9	79.3	**140.2**	**2.3**
	320	104.1	84.2	**188.3**	**3.4**
DAX	320	–	44.1	–	–

Table 4: Average Tencent Sort duration breakdown.

coalesce the temporary log write without losing consistency. Assise-Opt achieves 2.1× higher throughput than Assise. Fileserver has few redundant writes and Assise-Opt is only 7% faster.

MinuteSort. We implement and evaluate Tencent Sort [46], the current winner of the MinuteSort external sorting competition [7]. Tencent Sort sorts a partitioned input dataset, stored on a number of cluster nodes, to a partitioned output dataset on the same nodes. It conducts a distributed sort consisting of 1) a range partition and 2) a mergesort (cf. MapReduce [32]). Step 1 presorts unsorted input files into ranges, stored in partitioned temporary files on destination machines. Step 2 reads these files, sorts their contents, and writes the output partitions. Each step uses one process per partition; the parallelism equals the number of partitions. A distributed file system stores the input, output, and temporary files, implicitly taking care of all network operations.

We benchmark the MinuteSort Indy category, which requires sorting a synthetic dataset of 100B records with 10B keys, distributed uniformly at random. Creating a 2GB input partition per process, we run 160 or 320 processes in parallel, uniformly distributed over 4 machines. MinuteSort does not require replication, so we turn it off. It calls `fsync` only once for each output partition, after the partition is written. We compare a version running a single Assise file system with one leveraging per-machine NFS mounts. For Assise, we configure the temporary and output directories to be colocated with the mergesort processes. We do the same for NFS, by exporting corresponding directories from each mergesort node. We conduct three runs of each configuration and report the average. We use the official competition tools [7] to generate and verify the input and output datasets. We use equal dataset sizes to compare Assise and NFS, rather than equal time. Table 4 shows that Assise sorts up to 2.2× faster than NFS. Running twice the number of processes only marginally improves performance, as Assise is bottlenecked by network bandwidth.

To show that Assise's POSIX implementation does not reduce performance, we modify the sort step to map all files into memory using EXT4-DAX and use processor loads and non-temporal stores to sort directly in NVM, rather than using file IO. We can see that the sort phase is 3% slower with DAX. libc buffers IO in DRAM to write 4KB at a time to NVM, performing better than direct, interleaved appends of 100B records.

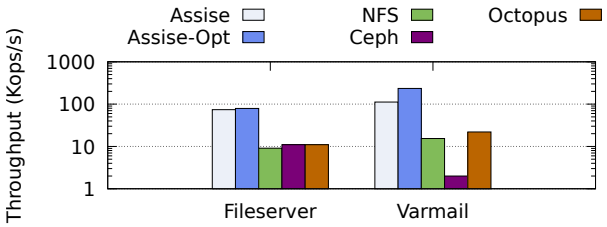


Figure 8: Avg. Varmail and Fileserver throughput. Log scale.

5.4 Availability

Ceph and Assise are fault tolerant. We evaluate how quickly these file systems return an application back to full performance after the fail-over and recovery situations of §3.4. To do so, we run LevelDB on the same dataset (§5.3) with a 1:1 read-write ratio and measure operation latency before, during, and after fail-over and recovery. We report average results over 5 benchmark runs.

Process fail-over. For this benchmark, we simply kill LevelDB. In this case, the failure is immediately detected by the local OS and LevelDB is restarted. Ceph can reuse the shared kernel buffer cache in DRAM, resulting in LevelDB restoring its database after 1.63s and returning to full performance after an additional 2.15s, for an aggregate 3.78s fail-over duration. With Assise, the DB is restored in 0.71s, including recovery of the log of the failed process and reacquisition of all leases. Full-performance operations occur after an additional 0.16s, for an aggregate 0.87s fail-over time. Assise recovers this case 4.34× faster than Ceph, showing that process-local caches do not impede fast recovery.

OS fail-over. NVM's performance allows for instant local recovery of an OS failure, rather than requiring a backup replica. To demonstrate, we run the primary in a virtual machine (VM). We kill the primary VM, then immediately start a new VM from a snapshot stored in NVM. The snapshot starts in 1.66s. We restart SharedFS within the new VM, which recovers the file system within 0.23s. Finally, as in the process fail-over experiment, LevelDB is restarted and resumes database operations after another 0.68s. The aggregate fail-over time is 2.57s, 40× faster than Ceph's fail-over to a backup replica (evaluated next).

Fail-over to hot backup. All following experiments use 2 machines (primary and backup). The LevelDB client processes poll the file system's cluster manager for membership state, using a standard primary-backup ZooKeeper design pattern for node fail-over [47]. LevelDB initially runs on the primary, where we inject failures. Failures are detected by LevelDB clients using a 1s heartbeat timeout via the cluster manager. Once a node failure is detected, LevelDB immediately restarts on the backup.

A time series of measured LevelDB operation latencies during one experiment run is shown in Figure 9. Pre-failure, we see bursts of low latency in between stretches of higher latency. This is LevelDB's steady-state. Bursts show LevelDB

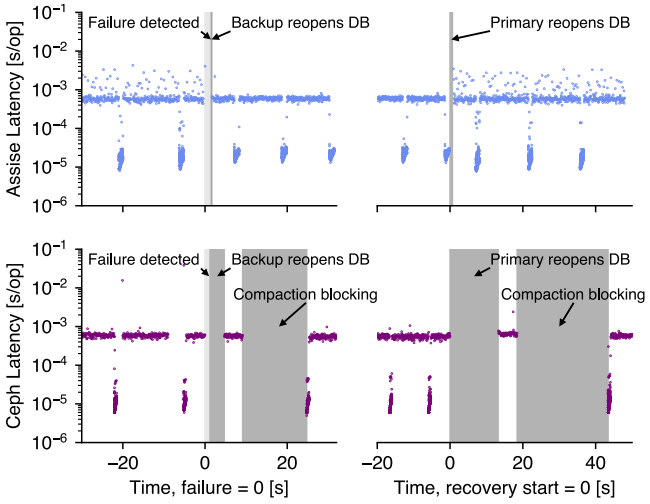


Figure 9: LevelDB operation latency time series during fail-over and recovery. Log scale.

writes to its own DRAM log. These are periodically merged with files when the DRAM log is full, causing writes that are higher latency (and sometimes blocking with Ceph), as the writes wait on the log to become available.

We inject a primary failure by killing the primary's file system daemon (SharedFS for Assise and OSD for Ceph) and LevelDB. During primary failure, no operations are executed. It takes 1s to detect the failure and restart LevelDB on the backup (light shaded box). Due to unclean shutdown, LevelDB first checks its dataset for integrity before executing further operations (dark shaded box). For failover, Assise need only evict the per-process log (up to 1GB) on the backup hot replica, making fail-over near-instantaneous. LevelDB returns to full performance in both latency and throughput 230ms after failure detection. Ceph takes 3.7s after failure detection to return to full performance. However, LevelDB stalls soon thereafter upon compaction (further dark shaded box), which involves access to further files, resulting in an additional 15.6s delay, before reaching steady-state. Ceph's long aggregate fail-over time of 23.7s is due to Ceph losing its DRAM cache, which it rebuilds during LevelDB restart. Assise reaches full performance after failure detection 103× faster than Ceph. LevelDB performs better on the backup, as neither file system has to replicate.

Primary recovery. After 30s, we restart the file system daemons on the primary, emulating the time for a machine reboot from NVM. During this time, many file system operations occur on the backup that need to be replayed on the primary. As soon as the primary is back online, we cleanly close the database on the backup and restart on the primary. Both Assise and Ceph allow applications to operate during primary recovery, but performance is affected. Assise detects outdated files via epochs and reads their contents from the remote hot replica upon access. Once read, the local copy is updated, causing future reads to be local. LevelDB returns to full performance 938ms after restarting it on the recovering primary.

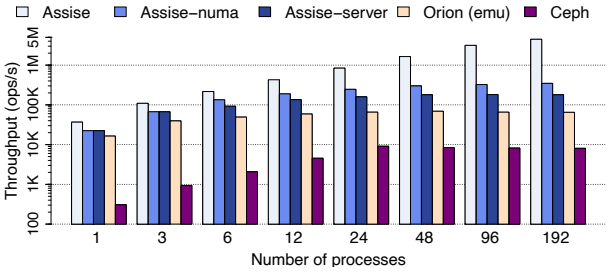


Figure 10: Scalability of atomic 4KB file operations. Log scale.

Ceph also rebuilds the local OSD, but eagerly. Ceph takes 13.2s before LevelDB serves its first operation due to contention with OSD recovery and suffers another delay of 24.9s on first compaction, reaching full performance 43.4s after recovery start. Assise recovers to full performance 46× faster than Ceph.

Fail-over to cold backup. We measure cascaded LevelDB fail-over time to an Assise replica with a cold cache. LevelDB serves its first request on the cold backup 303ms after failure detection, but with higher latency due to SSD reads. LevelDB returns to full performance after another 2.5s. At this point, the entire dataset has migrated back to cache.

5.5 Scalability

We evaluate Assise's scalability via 1) sharded file operations under increasing load and increasingly localized lease management, and 2) parallel email delivery in Postfix [79].

5.5.1 Sharded File Operations

Processes in parallel create, write, and rename 4KB files with random data in private directories. This benchmark uses 3 machines (6 sockets) and can scale throughput linearly with the number of processes. To eliminate network bottlenecks to scalability, we turn replication off.[1] Figure 10 presents average throughput over 5 runs of an increasing number of processes, each operating on 480K files, balanced over processor sockets. Ceph uses 3 sharded MDSes (1 per machine). However, MDS sharding has negligible impact on Ceph's performance.

Ceph's remote MDSes have high overhead for atomic operations, as each client has to communicate with remote MDSes. This design prevents scalability beyond 8Kops/s. We emulate Orion by restricting CC-NVM to use a single SharedFS lease manager. In this case, data is stored on local NVM, but atomic operations still use a remote lease manager. Orion has RDMA mechanisms that simplify communicating with its MDS, but these mechanisms cannot be used for operations that affect multiple inodes (e.g., renames). Orion and Assise both use RDMA RPCs. While Orion operates in the kernel, our emulation uses user-level RDMA, which is light-weight, and Orion (emu) outperforms Ceph by 8×.

[1] Due to the small size of our cluster, primary nodes would replicate to each other and scalability would be limited by per-node link bandwidth. A larger cluster would replicate to independent machines for each primary.

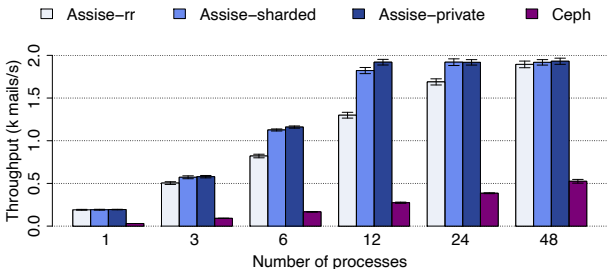


Figure 11: Postfix mail delivery throughput scalability.

To break down the benefit of local lease management in Assise, we progressively shard it, first by server (Assise-server), then by socket (Assise-numa), and finally by process (Assise). Assise-server outperforms Orion (emu) by 2.77× and Assise-numa improves throughput by another 1.93×. Assise scales linearly with the number of processes until it hits NVM write bandwidth, improving throughput by another 12.86×. Assise outperforms Orion by 69× and Ceph by 554× at scale.

5.5.2 Postfix

We use the unmodified Postfix mail server to measure the performance of parallel mail delivery. A load balancer machine forwards incoming email from as many client machines as necessary to maximize throughput to Postfix queue daemons running on 3 testbed machines, configured as replicas. On each Postfix machine, a pool of delivery processes pull email from the machine-local incoming mail queue and deliver it to user Maildir directories on a cluster-shared distributed file system. To ensure atomic mail delivery, a Postfix delivery process writes each incoming email to a new file in a process-private directory and then renames this file to the recipient's Maildir.

We send 80K emails from the Enron dataset [51], with each email reaching an average of 4.5 recipients. This results in a total of 360K email deliveries. Each email has an average size of 200KB (including attachments) and the dataset occupies 70GB. We repeat each experiment 3 times and report average mail delivery throughput and standard deviation (error bars) in Figure 11 over an increasing number of delivery processes, balanced over machines. We compare various Assise configurations and Ceph with 2 MDSes (1 and 3 MDSes performed similarly).

Round-robin. In the first configuration (Assise-rr) the load balancer uses a round-robin policy to send emails to mail queues. Due to a lack of locality, mails delivered to the same Maildir often require synchronization across machines, causing CC-NVM to frequently delegate leases remotely, which increases delivery latencies. Despite this, Assise-rr is able to outperform Ceph by up to 5.6× at scale. Ceph cannot improve throughput much further—even with 300 delivery processes, its throughput improves by 8% versus 48 processes.

Sharded. We shard Maildirs by Enron suborganization over machines [26]. The load balancer is configured to prefer the recipient's shard. For mail messages with multiple recipients, it picks the shard with the most receivers. In case of mail queue overload, the load balancer sends mail to a random unloaded shard. Sharding users in this manner provides up to 20% better performance (Assise-sharded) due to the fact that repeated deliveries to users of the same clique are likely to occur on the same server, allowing CC-NVM to synchronize delivery locally. At 15 processes, we are network-bound due to replication. Sharding did not improve Ceph's performance.

Private directories. We shard Maildirs by delivery process, using process IDs for Maildir subdirectories, thereby eliminating the need for synchronization (Assise-private). This change is not backward compatible with existing mail readers, but it is the logical limit for sharding-based optimization. Assise-private scales linearly until it is bottlenecked by network bandwidth, but performance is similar to Assise-sharded. This shows that local synchronization in Assise has minimal overhead. Ceph performance continues to be gated by the MDS.

Summary. Our results show that, with careful sharding of the workload, Assise's hierarchical coherence allows LibFS processes to synchronize deliveries locally, providing almost the same performance and scalability as private directories.

6 Conclusion

Assise is a distributed file system that provides low tail latency, high throughput, scalability, and high availability with a strong consistency model. To take advantage of low-latency NVM, Assise demonstrates that filesystem metadata and data should be colocated with applications. Colocation not only enables high performance, but also fast recovery. Assise proposes a novel, crash-consistent cache coherence protocol that can leverage the performance of NVM, while providing linearizability. Assise uses hot replicas in NVM to minimize application recovery time and ensure data availability, while leveraging a crash-consistent file system cache-coherence layer (CC-NVM) to provide scalability. In comparing with several state-of-the-art file systems, our results show that Assise improves write latency up to 22×, throughput up to 56×, fail-over time up to 103×, and scalability up to 6× versus Ceph, while providing stronger consistency semantics.

Assise is available at https://github.com/ut-osa/assise.

Acknowledgments. Waleed Reda was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC), funded by the European Commission (EACEA) (FPA 2012-0030). This work is supported in part by ERC grant 770889, NSF grant CNS-1900457, and the Texas Systems Research Consortium. This work is also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2020R1C1C1014940). We thank Intel for access to the evaluation testbed. We thank the anonymous reviewers and our shepherd, Kim Keeton, for their comments and feedback.

References

[1] Amazon Elastic Block Store (EBS). https://aws.amazon.com/ebs/.

[2] Amazon Elastic File System (EFS). https://aws.amazon.com/efs/.

[3] Amazon S3. https://aws.amazon.com/s3/.

[4] Apache Crail. http://crail.apache.org/.

[5] Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[6] Octopus - github repository. https://github.com/thustorage/octopus.

[7] Sort benchmark home page. http://sortbenchmark.org/.

[8] syscall_intercept. https://github.com/pmem/syscall_intercept.

[9] Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, Sept. 2014.

[10] Apache ZooKeeper. https://zookeeper.apache.org, Aug. 2017.

[11] Persistent memory programming, Aug. 2017. http://pmem.io/.

[12] The Sprite Operating System. https://www2.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html, Aug. 2017.

[13] Accelio, Aug. 2018. https://github.com/accelio/accelio.

[14] Intel Optane DC persistent memory, Mar. 2019. http://www.intel.com/optanedcpersistentmemory.

[15] Intel SSD DC P4610 1.6TB, Apr. 2019. Google Shopping search. Lowest non-discount price.

[16] NFS - Xfstests, 2019. http://wiki.linux-nfs.org/wiki/index.php?title=Xfstests&oldid=5652.

[17] NVM Express over Fabrics 1.1, 2019. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf.

[18] Xfstests, 2019. https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/.

[19] DDR4-3200 DRAM ECC Registered 128GB, Oct. 2020. Google Shopping search. Lowest non-discount price.

[20] Intel Optane DC Persistent Memory Module 128GB, Oct. 2020. Google Shopping search. Lowest non-discount price.

[21] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of Ceph evolution. In *27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[22] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, 2008.

[23] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 109–126, 1995.

[24] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, 2009.

[25] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen. Failure analysis of virtual and physical machines: Patterns, causes and characteristics. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 1–12, 2014.

[26] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. In *Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 178–179, 1981.

[27] M. Burrows. *Efficient data sharing*. PhD thesis, University of Cambridge, UK, 1988.

[28] Ceph Documentation. Differences from POSIX. http://docs.ceph.com/docs/master/cephfs/posix/.

[29] Y. Chen, Y. Lu, and J. Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *14th EuroSys Conference 2019*, pages 1–14, 2019.

[30] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, 2013.

[31] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *10th USENIX Conference on File and Storage Technologies*, FAST'12, 2012.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation*, OSDI'04, 2004.

[33] J. Dean and S. Ghemawat. LevelDB: A Fast Persistent Key-Value Store. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html, 2011.

[34] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen. Performance and protection in the ZoFS user-space NVM file system. In *27th ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[36] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 61–74, 2010.

[37] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, 2003.

[38] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, 1989.

[39] T. Haynes and D. Noveck. Network file system (NFS) version 4 protocol, Mar. 2015. https://tools.ietf.org/html/rfc7530.

[40] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th edition, 2017.

[41] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter 1994 Technical Conference*, WTEC'94, 1994.

[42] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. West. Scale and performance in a distributed file system. *SIGOPS Oper. Syst. Rev.*, 21(5):1–2, Nov. 1987.

[43] InsideHPC. Intel Optane DC persistent memory comes to Oracle Exadata X8M, Sept. 2019. https://insidehpc.com/2019/09/intel-optane-dc-persistent-memory-comes-to-oracle-exadata-x8m/.

[44] N. S. Islam, M. Wasi-ur Rahman, X. Lu, and D. K. Panda. High performance design for HDFS with byte-addressability of NVM and RDMA. In *2016 International Conference on Supercomputing*, ICS '16, pages 8:1–8:14, 2016.

[45] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the Intel Optane DC Persistent Memory Module, Apr. 2019. https://arxiv.org/abs/1903.05714v2.

[46] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub. Tencent sort. Technical report, Tencent Corporation, 2016. http://sortbenchmark.org/TencentSort2016.pdf.

[47] F. Junqueira and B. Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, Inc., 1st edition, 2013.

[48] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, 2019.

[49] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

[50] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 297–312, 2018.

[51] B. Klimt and Y. Yang. The Enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, pages 217–226. Springer, 2004.

[52] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, 2017.

[53] T. Le, J. Stern, and S. Briscoe. Fast memcpy with SPDK and Intel I/OAT DMA engine, Apr. 2017. https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.

[54] S. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. RECIPE: Reusing concurrent in-memory indexes for persistent memory. In *27th ACM Symposium on Operating Systems Principles*, 2019.

[55] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, 2014.

[56] H. Li, Y. Zhang, D. Li, Z. Zhang, S. Liu, P. Huang, Z. Qin, K. Chen, and Y. Xiong. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 15:1–15:17, 2019.

[57] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[58] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: An RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 773–785, 2017.

[59] P. MacArthur and R. D. Russell. A performance study to guide RDMA programming decisions. In *IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 778–785, 2012.

[60] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Linux Symposium*, volume 2, June 2007.

[61] Mellanox. Mellanox Introduces Revolutionary DPU based SmartNICs for Making Secure Cloud Possible, 2019. https://blog.mellanox.com/2019/08/mellanox-introduces-revolutionary-smartnics-for-making-secure-cloud-possible/.

[62] C. Metz. The epic story of Dropbox's exodus from the Amazon cloud empire, Mar. 2016. https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/.

[63] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 257–273, 2014.

[64] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[65] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Crashmonkey and ACE: Systematically testing file-system crash consistency. *ACM Trans. Storage*, 15(2), Apr. 2019.

[66] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[67] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 433–448, 2014.

[68] H. Qiu, X. Wang, T. Jin, Z. Qian, B. Ye, B. Tang, W. Li, and S. Lu. Toward effective and fair RDMA resource sharing. In *2nd Asia-Pacific Workshop on Networking*, pages 8–14, 2018.

[69] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[70] A. Rosenbaum and A. Margolin. Dynamically-Connected Transport, 2018. Talk. 14th Annual Open Fabrics Alliance Workshop.

[71] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 69–87, 2018.

[72] SNIA. *NVM Programming Model (NPM) Version 1.2*, June 2017.

[73] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, July 1981.

[74] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and atomic RDMA-based replication. In *2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 851–863, 2018.

[75] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX ;login:*, 41(1), 2016.

[76] S.-Y. Tsai, Y. Shan, and Y. Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, pages 33–48, 2020.

[77] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating Systems Design and Implementation*, OSDI'04, 2004.

[78] B. K. R. Vangoor, V. Tarasov, and E. Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 59–72, 2017.

[79] W. Venema. Postfix project. http://www.postfix.org/.

[80] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. In *10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 357–370, 2013.

[81] S. Watanabe. *Solaris 10 ZFS Essentials*. Prentice Hall Press, USA, 1st edition, 2010.

[82] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, 2006.

[83] J. Xu, J. Kim, A. Memaripour, and S. Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 427–439, 2019.

[84] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies*, FAST '16, pages 323–338, 2016.

[85] J. Yang, J. Izraelevitz, and S. Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies*, FAST '19, pages 221–234, 2019.

[86] ZDNet. Google cloud taps new Intel memory module for SAP HANA workloads, July 2018. https://www.zdnet.com/article/google-cloud-taps-new-intel-memory-module-for-sap-hana-workloads/.

[87] ZDNet. Baidu swaps DRAM for Optane to power in-memory database, Aug. 2019. https://www.zdnet.com/article/baidu-swaps-dram-for-optane-to-power-in-memory-database/.

[88] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 461–476, 2018.

Persistent State Machines for Recoverable In-memory Storage Systems with NVRam

Wen Zhang
UC Berkeley

Scott Shenker
UC Berkeley/ICSI

Irene Zhang
Microsoft Research/University of Washington

Abstract

Distributed in-memory storage systems are crucial for meeting the low latency requirements of modern datacenter services. However, they lose all state on failure, so recovery is expensive and data loss is always a risk. Persistent memory (PM) offers the possibility of building fast, persistent in-memory storage; however, existing PM systems are built from scratch or require heavy modification of existing systems. To rectify these problems, this paper presents Persimmon, a PM-based system that converts existing distributed in-memory storage systems into persistent, crash-consistent versions with low overhead and minimal code changes.

1 Introduction

In the past decade, distributed in-memory storage systems have become ubiquitous. Facebook and Twitter have petabytes of in-memory storage [2, 75], and in-memory replicated systems such as NOPaxos [58] and TAPIR [113] can process transactions within microseconds while providing consistency and fault-tolerance. As datacenter networks become faster and kernel bypass removes OS bottlenecks, only in-memory storage systems will be able to keep up with network speeds.

Unfortunately, in-memory storage systems have a crucial drawback: their lack of durability means that failed nodes must recover from a replica or another source (e.g., a persistent back-end database), which can be extremely slow. For example, a Facebook memcached cluster can take hours to regain full capacity if repopulated from another "warm" cluster, or days if repopulated from backend storage [75]. Even worse, state can be permanently lost if all replicas crash such as in a full datacenter failure. To reduce the impact of failures, many popular in-memory systems (e.g., Redis [85], RAMCloud [76]) support persistence, but it requires additional, complex code and/or incurs high performance overhead.

Persistent memory (PM) offers a promising solution for in-memory services. It is durable, offers performance close to DRAM, and is increasingly available in large sizes. However, PM systems require crash consistency [7, 57, 71, 79] (i.e., no system invariants are violated on a crash), which is complicated and expensive to enforce. Maintaining crash consistency requires that operations are failure-atomic [45]; for example, on crashes, an operation's deallocations and pointer updates must either atomically succeed or fail to avoid violating the invariant that pointers do not point to deallocated memory.

To ensure failure atomicity, PM systems must carefully flush volatile CPU state at specific times and possibly use write-ahead logging or other techniques to correctly recover from failures. These added flushes and writes impose significant overhead. As a result, most existing PM storage systems are carefully written from scratch for correctness and performance; even then, none can achieve the performance of today's in-memory systems. Recent work, like RECIPE [57] and MOD [39], aim to reduce application complexity by converting existing data structures to persistence on PM; however, because they exploit certain data structure properties (e.g., non-blocking synchronization), they are not suited to all in-memory storage systems.

This paper aims to let existing in-memory storage systems more easily reap the benefits of persistent memory. We make the key observation that distributed systems are typically designed as RPC-processing state machines. State machines are an ideal abstraction for PM because: (1) they encapsulate application state for recovery; (2) their operations offer clear failure-atomic regions; and (3) their state can be recreated at any time by re-executing operations.

Based on this insight, we present Persimmon, a PM-based system that converts existing in-memory distributed storage systems into durable, crash-consistent versions with low overhead and minimal code changes. Persimmon offers a new abstraction for building PM applications: *persistent state machines* (PSM). PSMs offer a simple guarantee: once an operation on the PSM returns, its side-effects on the PSM will never be lost. PSM operations are also failure-atomic: if the operation did not return before a crash, either the entire operation will be applied after the crash or none of it. PSMs can run arbitrary application code; however, like other state machines (e.g., replicated state machines), PSM operations must not have external dependencies (e.g., they cannot open file

descriptors), must be deterministic, and are executed sequentially. As a result, Persimmon does not support multi-threaded applications that apply operations concurrently.

To minimize the performance overhead of accessing PM on the request processing path, Persimmon keeps two state machine copies, one in DRAM and one in PM. When the application invokes a PSM operation, Persimmon first executes the operation on the DRAM copy. If the operation is read-only, Persimmon returns. If the operation is read-write, Persimmon persistently logs the operation before returning. This design limits the critical path to DRAM for read-only operations and one sequential write to PM for read-write operations; however, it requires both a DRAM and a PM state machine copy, which can be large for in-memory storage systems.

On failure, Persimmon can recover the PSM by replaying the persistent log. However, to minimize recovery time, Persimmon asynchronously keeps the persistent state machine snapshot in PM up-to-date. The state machine abstraction lets Persimmon update the PM snapshot with a *crash-consistent shadow execution* of each PSM operation, which is then removed from the log. This design is crucial for large in-memory storage systems that might have terabytes of data. To recover, Persimmon simply copies the PM snapshot to DRAM, processes the remaining persistent log, and restarts the application. Our design uses a background process, which runs on another CPU, to perform the shadow execution, trading off the use of a CPU for faster recovery times.

From this description, it is clear that Persimmon minimizes the overhead of persistence on the request processing path. However, to achieve reasonable recovery times, the crash-consistent shadow execution of the log must also be efficient, so the log does not grow too large. Most of the difficult technical challenges lie in optimizing this shadow execution, and we are not aware of similar work that addresses this particular issue. Note that despite these technical challenges, using a persistent log is preferable to checkpointing for large in-memory storage systems that might have terabytes of data.

We use Persimmon to persist two in-memory distributed systems: Redis and TAPIR [113]. We implement both systems on Linux and with kernel-bypass networking. We evaluate Persimmon on three servers with 3 TB of Intel® Optane™ DC Persistent Memory and found:

- On a 90% read-heavy YCSB workload, Persimmon incurs no discernible overhead to the latency and throughput of standard Redis; and near-zero latency overhead and 5% throughput overhead over kernel-bypass Redis.
- On the Retwis benchmark, Persimmon incurs no discernible latency overhead and 5%–8% throughput overhead for both standard and kernel-bypass TAPIR.

Furthermore, on gigabyte datasets, both Redis and TAPIR can recover within 15 s after a crash. Porting each application to Persimmon required less than 150 lines of code changes.

Although this paper mainly focuses on porting *existing* in-memory applications to PM, Persimmon also simplifies the development of *future* PM application. Even with high-level libraries, like Intel's PMDK [80], it remains difficult to write PM code that is both fast and correct. In contrast, our PSM abstraction lets programmers write state machine code targeting regular memory, then Persimmon automatically provides persistence while correctly maintaining crash consistency with low overhead. Persimmon thus offers a solution for developing new, high-performance persistent applications as easily as developing in-memory applications.

2 Persimmon Overview

This section gives an overview of Persimmon, including its design goals and API, and defines the requirements and guarantees of the persistent state machine model. Persimmon is designed for the x86-64 processor with Intel® Optane™ DC Persistent Memory [46, 108]. We assume an underlying POSIX-based OS due to Persimmon's use of fork; however, the design could be modified for other environments.

2.1 Design Goals

We identify three goals for Persimmon's design.

Minimal Application Changes. Existing in-memory storage systems are highly optimized for low latency in everything from data structures to memory allocators. To maintain these optimizations and reduce programmer effort, Persimmon's first goal is to minimize changes to existing application code.

Strong Guarantees. Reasoning about application state after a crash is difficult for PM applications [61, 62]. To simplify applications and ensure crash consistency, Persimmon's second goal is to provide strong and clear persistence guarantees.

Good Performance. In-memory storage systems must respond to requests within microseconds, so they cannot afford the high cost of existing persistence mechanisms (e.g., logging to disk). To provide persistence with PM, Persimmon's last goal is to impose less than a microsecond of latency overhead on in-memory systems with no persistence while also providing fast recovery times on the order of seconds.

2.2 Persimmon Persistent State Machine Model

Persimmon targets distributed systems deployed within a single datacenter that largely keep their state in memory and offer high-performance RPC processing. We assume the application state needed for recovery can be encapsulated in a *persistent state machine* (PSM) with the following properties:

- *Does not have external dependencies*. The state machine must contain no references to state outside the application process's address space; e.g., it cannot have file descriptors or open sockets.
- *Executes deterministically.* Each operation executes identically every time with no dependence on external inputs (e.g., the current time, random numbers) other than the operation arguments.
- *Has no external side-effects.* State machine operations must perform only computation and memory allocation and de-

Persimmon Interface

- `psm_init()` → `bool` - Initialization function; returns true if the application is in recovery.
- `psm_invoke_rw(op)` - Invoke read-write `op` with persistence on the state machine.
- `psm_invoke_ro(op)` - Invoke read-only `op` without persistence on the state machine.

Figure 1: Persistent state machine API implemented by Persimmon.

allocation (e.g., `mmap` and `munmap`). They must not invoke code with side-effects outside the application process (e.g., syscalls other than memory allocation).

These properties are common to state machine abstractions, and are required for correct shadow execution with Persimmon. Similar to replicated state machines (RSMs), persistent state machines require that operations execute sequentially for determinism. Due to the popularity of RSMs in the datacenter, we believe this requirement to be reasonable for many applications. For applications that require concurrency, it may be possible to apply existing techniques developed for RSMs [52, 72]; however, we defer the exploration of these techniques to future work.

2.3 Persimmon Persistent State Machine API

Persimmon provides a minimal application programming interface through its user-level library. The Persimmon library presents three functions to applications (Figure 1) that: (1) initialize the persistent state machine, (2) invoke a read-write PSM operation and (3) invoke a read-only operation. We offer the third function as an optimization for RPCs that only inspect state but do not update it, since many in-memory applications have a read-heavy workload. Programmers use the invocation functions to call existing application functions (e.g., execution of Redis commands on Redis data structures). Persimmon directly executes these functions on the PSM, so they must follow the properties laid out above.

An application starts by invoking the `psm_init` function, which returns a flag indicating whether the application has just recovered from a crash. If recovered, the persistent state machine will be returned to its state after the last completed operation. If not in recovery, the application should initialize the state machine by invoking an initialization operation (e.g., creating an empty Redis hash table) with `psm_invoke_rw`. The application can then begin RPC processing. On each RPC, we expect the application to invoke `psm_invoke_rw` if the RPC updates application state that is later needed for recovery. For correct recovery, the application must invoke the PSM operation *before* responding to the RPC. For RPCs that only access application state and do not make updates that must later be recovered (e.g., Redis `GET` operations), the application can use `psm_invoke_ro` for lower overhead.

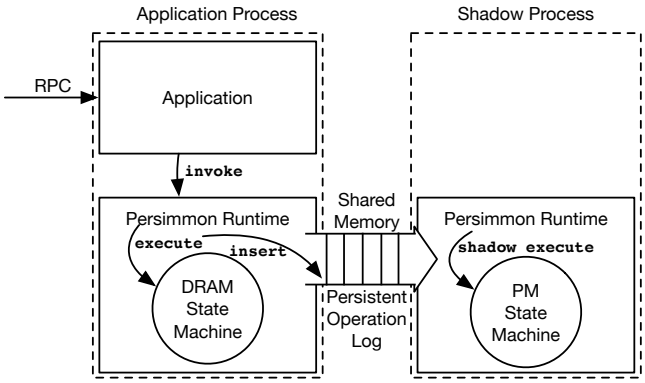


Figure 2: Persimmon runtime.

2.4 Persimmon Persistent State Machine Guarantees

Persimmon ensures three guarantees for invoked PSM operations. The first applies to all invoked operations, while the remaining two only apply to `psm_invoke_rw` operations.

- *Linearizability.* Persimmon guarantees that all state machine operations are run in a serial order and that serial order reflects the order in which operations are submitted to Persimmon [40].
- *Durability.* Persimmon guarantees that persistent, read-write state machine operations are never lost once they return, regardless of machine failures. Operations will never roll back and their state modifications are never lost.
- *Failure Atomicity.* Persimmon guarantees that state machine operations are failure-atomic. If the operation has not returned before failure, then on recovery, the state machine will reflect a state entirely before the operation has run or entirely after.

While these guarantees are simple, they are sufficient to build a crash consistent application in the face of failures. Because each state machine operation is failure-atomic, applications can easily maintain crash consistency by grouping updates to related data structures into a single operation and ensuring that no invariants are violated at the end of each operation.

3 Persimmon Runtime

Persimmon runs in two processes to support executing unmodified state machine code in DRAM and, for shadow execution, instrumented state machine code on PM. The *application process* runs the application and the DRAM state machine copy, while the *shadow process* runs the crash-consistent, shadow execution of the PM state machine copy. Persimmon's runtime shares the application process' address space with the rest of the application and completely owns the shadow process. Figure 2 summarizes Persimmon's runtime organization.

3.1 Data Structures

To ensure fast recovery and failure atomicity for the persistent state machine, Persimmon maintains two in-memory state machine snapshots and the data structures listed in Table 1.

Table 1: Data structures maintained by Persimmon.

Name	Persistent?	Data structure	Purpose	Elements	Operations
Operation log	Yes	Fixed-size queue	Records invoked operations	Serialized operations	`push`, `pop`
DRAM snapshot	No	—	To execute state machine operations on critical path	—	—
PM snapshot	Yes	—	Persists effects of operations	—	—
Region table	Yes	Resizable array	Records memory used by PM snapshot	$\langle addr, size, path \rangle$	`insert`, `remove`
Undo log	Yes	Resizable stack	Provides crash consistency for PM snapshot	Data entry: $\langle addr, size, data \rangle$ Commit entry: $\langle new_tail \rangle$	`append`, `clear`

We detail the snapshots and data structures below:

- *Operation log.* Persimmon records each invoked read-write operation in the operation log. The application and shadow processes use the operation log as a shared single producer, single consumer queue.
- *DRAM state machine snapshot.* Persimmon uses this snapshot to execute state machine operations on the critical path. It is always up-to-date and is used to serve all read-only operations without persistence.
- *PM state machine snapshot.* Persimmon asynchronously updates this snapshot using shadow execution. The snapshot is up-to-date up to the end of the log.
- *Region table.* Persimmon records memory allocated by the PM state machine snapshot. The PM snapshot is managed at the granularity of *PM regions*, each of which is a contiguous chunk of PM backed by a file.
- *Undo log.* Persimmon uses write-ahead logging for crash-consistent shadow execution. Persimmon instruments state machine code and record every overwritten memory value in the undo log to ensure that a partially executed state machine operation can be rolled back on recovery.

As Persimmon processes state machine operations, it appends them to the operation log while the shadow process digests the log by re-executing each operation on the PM snapshot. Operations in the log represent how far the PM snapshot lags behind the DRAM snapshot. On recovery, operations in the log must be re-executed on the PM snapshot before the application can restart. We keep the log size below a fixed upper bound to ensure that the PM snapshot does not lag the DRAM snapshot by too much and require too much re-execution on recovery. Persimmon implements the operation log as a circular buffer with head and tail pointers, and assumes no operation's arguments are larger than the log size.

3.2 Initialization and Normal Execution

When the application calls `psm_init`, Persimmon initializes its runtime in the following way:

1. Allocate the operation log.
2. Start the shadow process.
3. Initialize the DRAM and PM state machine snapshots.
4. Initialize the region table with PM region metadata (§ 3.4).
5. Allocate the undo log as a persistent array of entries (§ 4.2).

As the application runs, it invokes state machine operations through Persimmon, which are recorded to the log and eventually applied to the PSM. For each operation invoked through `psm_invoke_rw`, Persimmon performs the following:

1. Executes the operation on the DRAM snapshot.
2. Persists the operation as an entry in the operation log;
3. If the operation log is full, blocks until the shadow process digests more operations, freeing up space in the log.
4. Asynchronously, the shadow process re-executes each operation in the log on the PM snapshot using crash-consistent shadow execution (§ 4).

For operations invoked with `psm_invoke_ro`, Persimmon skips Steps 2–4. Persimmon blocks the application if the operation log is full. This design limits recovery time but requires that the shadow execution not lag behind too much as the application runs state machine operations. As a result, if the application invokes too many read-write state machine operations at a time, Persimmon will slow application performance significantly. We explore this phenomenon in our evaluation (§ 7.2.1).

3.3 Persimmon Shadow Process

Persimmon uses a separate process to perform shadow execution (the "shadow process"), where it switches to a dynamically instrumented version of the application for running the persistent state machine. Persimmon uses this instrumented version to manage persistent memory and ensure failure atomicity, which we discuss in §§ 3.4 and 4.3, respectively.

During initialization, Persimmon creates the shadow process by using `fork` to create a copy of the application process. Immediately after forking, the shadow process checkpoints itself using an existing Linux process checkpointing tool. This checkpoint conveniently stores essential process state that is orthogonal to Persimmon's main functionality (e.g., the process ID), and serves as a "base image" on which Persimmon manages PM regions. This initialization must happen before the application sets up external dependencies (e.g., opens sockets) to avoid causing process checkpointing to fail.

After taking the checkpoint, Persimmon replaces the shadow process's address space with persistent memory by creating a PM region for each existing application memory region. Specifically, Persimmon iterates through the existing memory regions using Linux's `/proc/self/maps` interface. For each region, Persimmon writes its content to a new PM

file, `mmap`s the file into the address space *over* the existing region, and inserts an entry into the PM region table. We skip over read-only regions, which we assume will never become writable; the stack region, which we assume contains no persistent state (§ 4.3); and the operation log.

Finally, the shadow process begins shadow-executing state machine operations from the operation log. Although it executes the same state machine code as the application process does, the code is executed on the shadow process' persistent address space, and so any modifications to memory are reflected in the PM state machine snapshot. However, Persimmon cannot directly execute unmodified application code on PM because it is not failure atomic and allocates DRAM, not PM. Instead, the shadow process turns on dynamic instrumentation to capture memory allocation and writes in order to allocate PM and write to it in a failure-atomic manner.

3.4 Persistent Memory Management

To be able to recover the shadow process after a crash, Persimmon must manage its persistent memory and keep track of its metadata persistently. Persimmon manages the shadow process's persistent memory at the granularity of *PM regions*, each of which is contiguous range of persistent virtual memory. A PM region's content is stored in a file in a direct-access (DAX) file system [59] on persistent memory; the file is `mmap`'ed into the shadow process' address space, allowing access to PM. Persimmon uses a persistent *region table* to manage metadata for PM regions; each element in the table has the form $\langle addr, size, path \rangle$, denoting a PM region $[addr, addr + size)$ backed by a file located at *path*.

Persimmon keeps the region table in an immutable file in PM. Whenever the region table changes, Persimmon writes the entire updated table to a new file and removes the old. Any PM region files that are "orphaned" after a region table update are garbage-collected after the new region table is written. This mechanism provides failure atomicity for region table updates; although expensive, it is easy to implement and invoked only rarely. Because the table is small, Persimmon keeps a cached copy of it in DRAM.

Every time the shadow state machine allocates or frees memory, Persimmon must translate the operation to allocate or free PM regions instead. Persimmon uses dynamic instrumentation to intercept `mmap` and `munmap` system calls, which are typically made by the application's memory allocator.[1] Persimmon's PM management thus operates underneath the memory allocator and does not constrain the application to use one specific allocator.

Persimmon currently supports `mmap` calls that allocate anonymous memory with no address requirement / hint or backing file. For a `mmap` call of this type, Persimmon creates a PM region of the allocated length and transparently maps the PM region to the intended address. We currently don't distinguish among page protection bits and assume that all allocated pages have all permissions. Persimmon supports `munmap` calls that free a single PM region or a part thereof, updating the region table before letting the calls through.

[1] A third memory management system call is `brk`, which is not supported by our current implementation but can be similarly supported.

4 Crash-Consistent Shadow Execution

Persimmon's shadow execution uses dynamic instrumentation and undo logging to provide failure atomicity for state machine operations executing arbitrary application code. We chose dynamic binary instrumentation over static compiler instrumentation because application code often calls functions from dynamically linked external libraries (e.g., string functions in libc), which are only available in binary form at runtime. However, dynamic instrumentation comes with higher overhead than static, and a future direction is to improve instrumentation performance by combining static and dynamic instrumentation [96].

4.1 Overview

During shadow execution, Persimmon uses an *undo log* in PM to record the value at a persistent location before it is overwritten, similar to many prior systems [9, 16, 33, 81, 88, 99]. To support arbitrary application code in the PSM, Persimmon uses memory-level physical logging so that it can roll back an incomplete state machine operation at recovery time by copying back the previous memory values. The undo log is a sequence of entries, which come in two types:

- A *data entry* records the old value at a persistent location.
- A *commit entry* signifies that a state machine operation has finished; it contains a sole field `new_tail` recording what the operation log's tail pointer should advance to after the current operation is consumed.

The undo log supports append and clear operations. Each operation blocks until it persists.

In the shadow process, Persimmon instruments every write to a persistent location to append a data entry to the undo log before letting the write through. When a state machine operation completes, we *commit* the operation and remove it from the operation log following these steps:

1. Flush all previous writes and wait for them to persist.
2. Compute the updated tail pointer for the operation log and append a commit entry to the undo log.
3. Remove the operation from the log by advancing the tail pointer as computed.
4. Clear the undo log.

The recovery procedure either finishes committing the operation in progress according to the commit entry if one exists, or rolls it back (§ 5).

Undo logging dominates the shadow state machine's performance because (1) it could add additional work to every memory write, and (2) an undo log append must wait for persistence to PM, which is slow. Therefore, our design aims to reduce the number of undo log appends and the amount of extra code executed per application memory write.

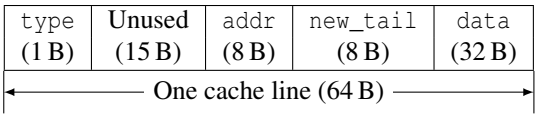


Figure 3: The layout of an undo log element (§ 4.2).

To achieve these goals, Persimmon logs at the granularity of aligned 32 B data blocks. Writes that straddle blocks will generate multiple undo log entries, and writes smaller than 32 B will result in at least 32 B of data being logged. This strategy is motivated by the observation that if a location has been undo logged, then any subsequent writes to that location need not be logged. By logging in larger blocks, Persimmon takes advantage of spatial locality in memory writes to coalesce logging for adjacent locations. Insisting that all blocks be equal-sized and aligned ensures that blocks never overlap and simplifies the detection of duplicate blocks (§ 4.3).

4.2 Undo Log Layout and Operations

The undo log consists of a persistent array A of fixed-size 64 B elements (which is the cache line size), and the undo log size n, which is stored in DRAM only. Our implementation fixes the array's total size to 2^{20} elements (32 MB), which is large enough for our applications. The array A is cache line-aligned, and so are its elements.

An array element either is *valid*, representing an undo log entry (§ 4.1), or is *invalid*. We maintain the invariant that $A[0..n-1]$ contains valid elements and $A[n..]$ contains invalid elements, so that n can be inferred from A upon recovery.

Figure 3 shows the layout of an array element. Each element is interpreted based on its `type` field:

- If `type` $= 0$, the element is invalid.
- If `type` $= 1$, the element is valid and represents a data entry that records the original value of $[\texttt{addr}, \texttt{addr}+32\,\text{B})$. The `addr` field must be a 32 B-aligned address.
- If `type` $= 2$, the element is valid and represents a commit entry with `new_tail` (§ 4.1).

When appending an entry, we make sure that the `type` field, which also indicates validity, persists no earlier than the other fields. This does not require using an extra persist barrier—since writes to the same cache line reach PM in program order [19, 84], we simply need to write the `type` field last.

To clear the log, we set `type` to zero for elements $A[0..n-1]$ from left to right. If a crash occurs during the clearing, the recovered process will see that $A[0]$ is invalid and will then clear the entire array again.

With this undo log organization, an append requires only one persist barrier (at the end), and consecutive appends write to PM sequentially and avoid repeatedly flushing a single cache line (which is known to incur high latency on Optane persistent memory [12, 92]). Although clearing the log at commit time requires writing to all n entries, it is rare due to the batch commit optimization (§ 4.3) and can be optimized by, e.g., maintaining a persistent commit sequence number.

4.3 Dynamic Binary Instrumentation

Persimmon dynamically instruments memory writes for undo logging. The bulk of the logging logic is implemented in the function `log_write(addr, sz)`, which rounds up the range $[\texttt{addr}, \texttt{addr}+\texttt{sz})$ to aligned 32 B blocks and appends an undo log entry for each block. Persimmon, in the shadow process, inserts a call to `log_write` before each application instruction that can write to memory. It translates repeat string operations into regular loops to instrument each iteration separately. For conditionally executed instructions, the instrumentation is executed only when the instruction is.

To minimize overhead from dynamic instrumentation and undo logging, Persimmon applies a number of optimizations:

Skipping the stack. Persimmon assumes that the application holds no persistent data on the stack. It does not save stack pages to PM, and as a result it does not need to instrument stack operations. Persimmon assumes that any memory operand that is an offset from the stack pointer `%rsp` points to the stack, and can thus efficiently skip instrumentation for a large number of instructions (notably, all `push`es and `pop`s). This assumption about `%rsp` usage can be validated at runtime by tracking all updates to the register [96], although we have not implemented this validation. The `log_write` function also skips writes to locations above `%rsp` minus 128 B (accounting for the red zone [63]), thereby filtering out any stack operations that do not use an offset from `%rsp`.

De-duplicating undo log entries. To avoid logging duplicate data, the `log_write` function maintains a hash set in DRAM for the `addr` field of existing undo log entries, and avoids appending entries whose `addr` already exists. This hash set must support lookup, insert, and clear operations, and fast lookup is key to making this optimization worthwhile.

Our hash set, closely modeled after the CPython dictionary [23], is implemented as a flat array of `addr`'s and resolves collisions with open addressing. The array size is fixed to 2^m (where $m = 14$ in our implementation); we use the simple hash function $h(\texttt{addr}) = \texttt{addr}/32$ (since `addr` is aligned to 32 B); and probing uses a linear recurrence with perturbation [23]. A zero element denotes an empty slot, and the hash set is cleared by zeroing out the entire array. Shadow execution commits once the hash set's load factor reaches 50% (see the batch commit optimization below); in case the hash set becomes full, the de-duplication optimization is disabled.

With this hash set implementation, lookups that do not encounter collision are extremely fast. This allows us to insert a fast path de-duplication check, which we detail next.

Fast-path de-duplication check. Although de-duplication in `log_write` avoids redundant logging, the function call before every write is still costly as it requires saving and restoring application registers. We therefore insert a "fast path" check *before* the function call to filter out easy-to-identify duplicates. For a memory write to address *dest* of size *sz*, the call to `log_write` is skipped if both conditions hold:

Listing 1: Instrumentation inserted before a memory write of *sz* bytes. `%dest` and `%tmp` are placeholders for any two distinct general-purpose 64-bit registers. The `.hash_array` label refers to the base address for the hash set array.

```
    (Reserve registers %dest and %tmp)
    (Compute destination of the write, store in %dest)
    (Reserve arithmetic flags)

    # First check: the write is contained in
    # a single block (not generated if sz=1).
    leaq    (sz-1)(%dest), %tmp
    xorq    %dest, %tmp
    cmpq    $31, %tmp
    ja      .slow_path

    # Second check: look in the hash set.
    movq    %dest, %tmp
    shrq    $2, %tmp
    andl    $131064, %tmp
    movq    .hash_array(%tmp), %tmp
    xorq    %dest, %tmp
    cmpq    $32, %tmp
    jb      .skip

.slow_path
    (Save application registers)
    (Call log_write)
    (Restore application registers)

.skip
    (Restore arithmetic flags)
    (Restore registers %dest and %tmp)
```

- The write is contained in a *single* aligned 32 B block, i.e.,
$$\lfloor dest/32 \rfloor = \lfloor (dest + sz - 1)/32 \rfloor.$$
- The block's address is found in the hash set on first try (without any probing), i.e., $H[i] = addr$ where H is the hash set array, $i = \lfloor dest/32 \rfloor \bmod 2^m$ according to the hash function, and $addr = \lfloor dest/32 \rfloor \times 32$ is the block address.[2]

Any memory write filtered out by the check is guaranteed to be a duplicate, and the check proves effective in our evaluation (§§ 7.2.1 and 7.2.3). Furthermore, as the computation required by the checks only require bit manipulations, the two checks can be implemented in 11 instructions using only two extra registers (one of which stores *dest* and is anyway required). Listing 1 shows the code inserted before a memory write.

Batch commit. Persimmon shadow-executes multiple state machine operations before committing, thus avoiding duplicate logging across multiple operations. After executing each operation, Persimmon checks to see if the de-duplication hash set is more than 50% full and if so, commits. We defer more intelligent batch sizing to future work.

Skipping newly allocated regions. Writes to a PM region that is newly allocated (i.e., after the most recent commit) do not need to be logged since, in case of a crash, the state machine will be reverted to before the region was allocated. The `log_write` function therefore maintains, in DRAM, a list of address ranges for newly allocated regions and searches through the list to skip logging such writes. This optimization is critical for supporting `calloc` implementations that manually zero out pages allocated with `mmap`.

5 Recovery

To minimize recovery times, recovery in Persimmon is relatively simple. After a crash, the first step is to restore the PM state machine snapshot to a consistent state:

- If the undo log contains a commit record, we set the operation log tail to *new_tail*. If an updated region table exists in PM, we switch to it and garbage collect the old region table. This completes the commit.
- If the undo log contains no commit record, we delete and garbage collect the updated region table (if one exists) and copy the old values from the undo log back to their respective locations. This rolls back the operation in progress at the time of the crash.

As a last step, we clear the undo log in both cases.

Starting from the consistent PM snapshot, Persimmon digests any remaining operations in the operation log so that all previously invoked operations are reflected in the snapshot. Replaying the log ensures that Persimmon maintains its guarantee that any invoked operation that returns will not be lost. This up-to-date snapshot is then copied into DRAM for the application process, and the application is restarted. Assuming that the PSM has captured all persistent application state, the application should be back in its pre-crashed state.

6 Implementation

We have implemented Persimmon in C++; it targets x86-64 Linux applications written in C or C++. We use CRIU [24] (v3.12) for process checkpointing during background process initialization (§ 3.3). For dynamic instrumentation (§ 4.3), we use DynamoRIO [8], a runtime code manipulation system. Persimmon's DynamoRIO client is linked into the application along with the DynamoRIO runtime. This setup allows Persimmon to start the application uninstrumented and only begin instrumentation in the background process once it is forked off. To avoid interfering with the application, our instrumentation code takes care not to call into shared libraries (e.g., libc), and instead uses DynamoRIO's memory allocator and our custom system call wrappers.[3]

7 Evaluation

Using our implementation, we demonstrate that Persimmon:

- Requires only a small amount of code modification for distributed in-memory storage systems.
- Achieves low overhead on workloads compared with no persistence for both Linux and kernel-bypass applications.
- Recovery quickly even for large memory sizes.

[2] Note that if no element exists in the hash set with hash value $h(addr)$, we have $H[i] = 0$ and the check fails as expected (assuming that the application never writes to the block starting at address zero).

[3] Because our DynamoRIO client is linked into the application, it is not loaded by DynamoRIO's private loader, which would have created a separate copy of each library used by the client.

Table 2: Rough lines of code changed to port Redis and TAPIR.

	Lines added / changed	
	Redis	TAPIR
Initialize Persimmon	7	10
Factor out state machine init.	36	34
Serialize state machine operation	26	12
Deserialize & execute operation	45	25
Check for read-only operations	1	1
Refactor for better performance	—	57
Total	115	139

We ported Redis, a popular key-value store, and TAPIR [113], a distributed transactional data store, to use Persimmon. We first describe the code changes required to port these applications (§ 7.1), followed by performance comparisons (§§ 7.2.1 and 7.2.2). Finally, we use microbenchmarks to evaluate the effectiveness of Persimmon's optimizations (§ 7.2.3).

7.1 Programming Experience

Although the Persimmon API is simple (Figure 1), porting real applications can require a few extra steps as real-world code bases are not always well-organized into a state machine abstraction, even if the application is processing RPCs. The programmer typically needs to:

1. Add a call to `psm_init()`, passing configuration arguments like the PM file system location.
2. Factor out the state machine initialization code into a single function, separating it from other initialization (e.g., network I/O), so that it can be skipped on recovery.
3. Write a function that serializes a state machine operation for operation logging. One can reuse the application's existing RPC serialization, but, for better performance, we found it valuable to use a custom format that is cheaper to parse in the shadow process.
4. Write a function that deserializes and executes an operation, to be invoked under instrumentation in Persimmon's shadow process. This function typically only needs to call the application's RPC handler using the deserialized operation, but may need to suppress any I/O by the handler (e.g., sending a reply over the network).
5. Insert checks to distinguish read-write operations from read-only ones; such checks likely already exist for applications that support state machine replication. Invoke Persimmon for read-write operations.

We performed all of these steps for Redis because it was not well-organized into a state machine, especially because it is written in C, which is not an object-oriented language. TAPIR, on the other hand, was is already designed as a state machine to work with its replication mechanism. Table 2 summarizes the code changes required to port Redis and TAPIR. We discuss each application in detail next.

7.1.1 Porting Redis

To port Redis, we treat each Redis command as a state machine operation and invoke it through Persimmon. To summarize, the changes made were:

1. Persimmon initialization took 6 lines of code (LoC).
2. To factor out the state machine initialization code, we separated out three blocks of code ($7+7+22$ lines) responsible for network and domain socket initialization, etc.
3. Redis operation serialization for the state machine took 26 lines of code. The serialization consists of the address of the Redis command's handler function[4] as well as the command's arguments.
4. Redis operation deserialization, parsing, and dispatch took 24 lines of code. Since executing Redis commands requires a "client", we reuse the fake client code from Redis AOF (21 lines), which also suppresses replies.
5. To determine whether an operation is read-only, we reused existing code from Redis' state machine replication.

In all, Persimmon allowed us to achieve persistence for Redis with roughly 100 lines of code changes. In contrast, Redis' own persistence implementation (AOF and RDB) consists of roughly 3000 lines of code, including complex logic such as request processing while log compaction is in progress. As another point of comparison, Pmem-Redis [82], a version of Redis that uses persistent memory, contains roughly 30 000 new lines of C code over Redis 4.0.0, although we note that Pmem-Redis contains features that are orthogonal to Persimmon (e.g., defragmentation). Xu et al. report that manually porting Redis to persistent memory using PMDK [80] "is not straightforward and requires large engineering effort" [104, § 3.3]. They list five difficulties, which include supporting the many different Redis objects with different encodings, carefully ordering writes to maintain crash consistency, etc. None of the difficulties arose when we ported Redis using Persimmon.

Despite minimal changes, our port is the most feature-complete PM Redis port that we know of. For example, Persimmon-Redis supports complex Redis data structures like sets and hashes while the persistent Redis from WHISPER [73,98] only supports simple key-value pairs, and Pmem-Redis lacks support for optimized encodings like `intset` and `zipmap`. We also support hash table resizing, which is not supported by P-Redis due to its complexity [104]. Persimmon-Redis supports these features "out of the box", without requiring additional code for each feature. In addition, Persimmon lets Redis retain jemalloc [47] as its memory allocator, which was carefully chosen by the Redis developers [86].

7.1.2 Porting TAPIR

The TAPIR transactional data store is built on top of the inconsistent replication (IR) protocol. We treat each IR RPC as a state machine operation. In the application, these RPCs

[4] As the shadow process is forked from the application process (§ 3.3), the application's code segment is mapped at the same location in both processes.

are serialized using Protocol Buffers [83], and a naive Persimmon port uses the same serialization for operation logging. Although easy to implement, this strategy causes significant performance degradation as Persimmon has to execute Protocol Buffer parsing under shadow execution, which is slow.

To improve performance, we refactored TAPIR's IR RPC handlers to take individual RPC fields as arguments, so that they can be called from the shadow process without first constructing protobuf objects. This refactoring involved roughly 50 LoC changes, which were mostly mechanical, and enabled operation logging using a simple custom format (like for Redis). In addition to this refectoring, we did the following:

1. Persimmon initialization took 11 lines of code.
2. To factor out state machine initialization, we moved two code blocks ($10+4$ lines) and modified ~20 LoC to allow creating a RPC handler without a network connection.
3. Operation serialization took 12 lines of code.
4. Operation deserialization and invocation took 17 lines of code, plus an extra 8 lines to suppress replies.
5. We reused application code to detect read-only operations.

The overall code change amounts to roughly 140 lines in total.

To recover from replica failures, TAPIR uses a complex in-memory recovery protocol, inspired by VR, which has been proven to be incorrect [69]—under certain failure conditions, TAPIR can lose operations when recovering, causing it to miss updates. Persimmon-TAPIR fixes this problem transparently and lets replicas correctly recover their state on failures.

One of the benefits of TAPIR is that replicas can recover and immediately begin processing transactions. However, without the most up-to-date state, these recovered replicas will degrade performance by serving stale reads and unnecessarily aborting writes. TAPIR particularly suffers from this performance degradation because it needs $\frac{3}{2}f+1$ to use the single-round trip fast path. For a 3-machine replica group, this number includes all of the participants. As a result, while the recovering replica is able to participate in transactions immediately, without the most up-to-date state, it is only hurting performance. However, Persimmon-TAPIR can significantly reduce this performance degradation by limiting the amount of state that the replica needs to recover.

7.2 Performance Evaluation

We evaluate Persimmon on three 52-core, dual-socket Intel Xeon Platinum 8272 2.6 GHz servers, each with 3 TB of Intel® Optane™ DC Persistent Memory in app direct mode and 768 GB of DRAM. We mount an ext4 file system in DAX mode [59] on the PM. Persimmon's application and background processes run on two physical cores on a single NUMA socket and use only DRAM, PM, and the NIC on that socket. To supply the client workload, we use a server with a 20-core dual-socket Xeon Silver 4114 2.2 GHz CPU, connected with with Mellanox CX-5 100 Gbps NICs and an Arista 7060CX 100 Gbps top-of-rack switch.

Table 3: Summary of Redis performance on YCSB (Zipfian constant = 0.75, 10% update, median of five runs). The persistence options are volatile ("Vol"), persistent through Persimmon ("PSM"), and persistent through append-only file ("AOF"), with our performance in bold. Shown are median latency at low load and peak throughput. Persimmon incurs, for Linux, negligible latency and throughput overhead and, for kernel bypass, negligible latency overhead and ~5% throughput overhead. Figure 4a shows the full latency vs throughput graph.

	Latency (µs)			**Throughput** (Kops)		
Redis Setup	Vol	**PSM**	AOF	Vol	**PSM**	AOF
Linux	17.8	**17.5**	18.6	227	**227**	107
Bypass	10.4	**11.2**	12.0	452	**429**	130

7.2.1 Redis Performance

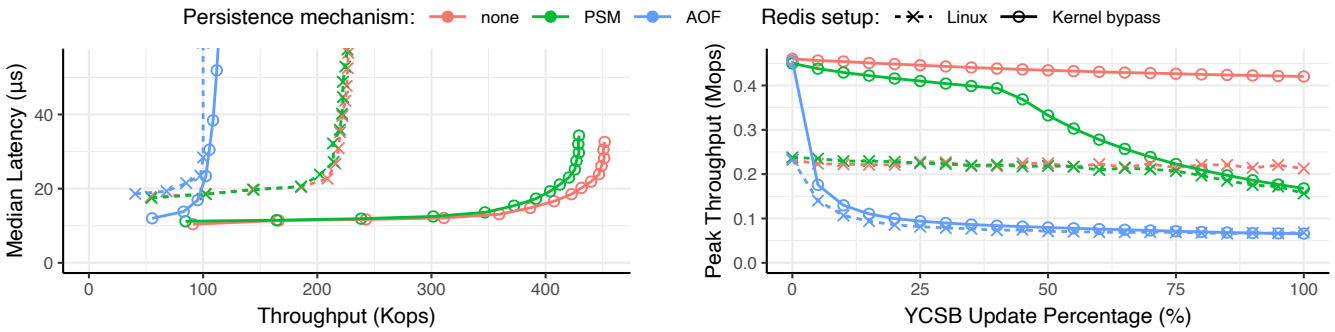
We use Persimmon to add persistence to two versions of Redis—regular Redis (v4.0.9), which processes requests over TCP using the POSIX API, and a high-performance, kernel-bypass version of Redis that uses the Demikernel's DPDK library OS [26, 112]. We compare both Persimmon-Redis versions to Redis's existing persistence mechanisms.

While Redis is an in-memory storage system, persisting Redis is popular enough for it to integrate two mechanisms for saving its state [85]: RDB, a snapshotting mechanism, and AOF, an append-only operation log. Using RDB requires pausing operation processing for a short period while Redis spawns a background process to checkpoint its database.

AOF logs every write operation received by the server to a file, similar to Persimmon's operation logging. However, Redis typically recommends only `fsync`'ing those logged operations periodically to avoid performance overhead [85], so operations can be lost on a crash. AOF must also avoid the operation log growing unboundedly, so it periodically creates an RDB snapshot, letting it truncate the log. This process further degrades performance, so the Redis developers recommend only snapshotting once or twice an hour [85].

In some sense, these mechanisms are orthogonal to Persimmon. They have features that Persimmon does not provide (e.g., compact and platform-independent serialization), but do not provide cheap, general-purpose persistence to in-memory storage systems. Their implementation is specific to Redis, requires significant implementation effort, and, as shown below, can cause large performance degradation.

Experiment setup. We evaluate Redis performance using the YCSB benchmark [21]. We implemented a custom multi-threaded YCSB client, using Shenango [78], which supports both TCP and Demikernel's UDP-based protocol. Following the official YCSB implementation [111], our client is closed-loop and does not use Redis pipelining, each YCSB record is represented with a Redis hash, and fields are accessed using Redis' `HSET`/`HGET` commands. We load 13 million records; as is YCSB default, each record has 10 fields (i.e., 130 million items in total), and each field has a 100 B value. Our client is-



(a) *Latency vs throughput for a 10%-update workload.* Persimmon incurs negligible overhead over the Linux baseline, and a roughly 5% overhead to the peak throughput over the kernel-bypass baseline.

(b) *Peak throughput vs update percentage.* Persimmon throughput stays within 9% of the kernel bypass baseline for up to 40%-update, and within 4% of the Linux baseline for up to 75%-update.

Figure 4: Redis performance on the YCSB benchmark (median of five runs, Zipfian constant = 0.75).

sues reads and updates according to a fixed ratio, and chooses which records to access according to a Zipfian distribution.

On the server side, the regular Redis server uses jemalloc (as is recommended for Linux) and our kernel-bypass Redis uses the Hoard memory allocator [3] (following the Demikernel implementation [26]). Where Redis AOF is enabled, we place the AOF log file on the PM file system, disable AOF rewriting, and configure it to always `fsync` before sending replies, providing the same level of durability as Persimmon.

End-to-end performance. We start with the end-to-end performance for a typical YCSB workload with 10% updates and a Zipfian constant of 0.75. We measure the latency and throughput for unmodified Redis, Persimmon Redis, and Redis AOF. Table 3 reports performance both on Linux and with kernel-bypass enabled through the Demikernel and shows:

- On Linux, Persimmon provides persistence while incurring negligible latency or throughput overhead.
- With kernel bypass, Persimmon incurs negligible latency overhead at low load, and a 5% degradation to peak throughput. Kernel bypass makes Redis significantly more efficient, so Persimmon has slightly more impact on performance.
- On Linux, AOF incurs < 1 µs latency cost but a $2\times$ throughput penalty, a much higher overhead than Persimmon.
- With kernel bypass, AOF again incurs a small latency overhead but a $3.5\times$ throughput loss.

Some of the overhead of AOF is likely due to inefficiencies in accessing PM through ext4 and could be reduced using a specialized PM file system like NOVA [104, 105] or SplitFS [48]. Overall, Persimmon offers persistence at a much lower cost than Redis's own custom persistence mechanism both on Linux and for future kernel-bypass deployments.

Figure 4a shows the full latency vs throughput plot for this workload with varied numbers of closed-loop clients. (Lower and to the right is better. The knee where latency goes up shows the peak throughput.)

Table 4: Redis recovery time and storage size for the three persistence mechanisms (median of three runs). Persimmon recovers 4.6×–6× faster than AOF and RDB. The discrepancy between the Linux and kernel bypass implementations is likely due to memory allocators differences (jemalloc vs Hoard).

	Recovery Time (s)			**Storage Size** (GB)		
Redis Setup	**PSM**	AOF	RDB	**PSM**	AOF	RDB
Linux	**14.6**	87.4	87.8	**23**	16	2.8
Bypass	**20.4**	93.3	93.5	**33**	16	2.8

Peak throughput vs update ratio. Since Persimmon only logs and shadow-executes write operations, its overhead depends on the workload's update-to-read ratio. Figure 4b shows peak Redis throughput as we vary the workload's update percentage. Persimmon's throughput remains within 4% of the baseline for up to 75%-update on Linux (represented by the red and green "×" lines in the middle of the graph), and within 9% of the baseline for up to 40%-update for kernel bypass (represented by the red and green "○" lines at the top). After these points, the shadow state machine becomes saturated and the throughput drops precipitously. Both versions can easily handle read-heavy workloads, which are common in practice.

Recovery speed and storage size. Table 4 shows Redis' recovery speed and storage usage under different persistence mechanisms if we kill Redis after loading our YCSB dataset.[5] Because Persimmon persists application data in its in-memory format, the bulk of its recovery is physically copying PM regions back into DRAM, while AOF and RDB require reloading the database. Consequently, Persimmon recovery is faster by 4.6× (on Linux) to 6.0× (for kernel bypass) and, we believe, can be further optimized by copying PM regions in parallel using multiple cores. However, Persimmon's space

[5]The Persimmon recovery measurements do not include operation replay because it would take negligible time—since the operation log is only 32 MB, replaying even a full log would only take roughly one second for YCSB.

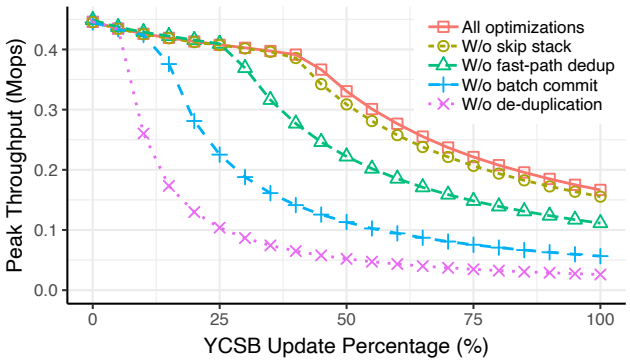


Figure 5: Disabling each optimization degrades Redis throughput on YCSB (Zipf = 0.75, kernel bypass, median of five runs). Note that disabling de-duplication also disables the fast-path check.

usage tracks the application's RAM usage while AOF and RDB can use more compact serialization formats. The discrepancy between Linux and kernel-bypass Redis is most likely because they use different memory allocators (jemalloc vs Hoard), which lead to differing amounts of memory consumption and exhibit different allocation performance.

Effectiveness of optimizations. To evaluate Persimmon's optimizations (§ 4.3), we measure Redis performance under Persimmon after disabling each optimization separately (note that disabling undo log de-duplication also disables the fast-path de-duplication check).[6] Figure 5 shows that no optimization can be removed without degrading performance.

7.2.2 TAPIR Performance

As with Redis, we use Persimmon to add persistence to two versions of TAPIR—regular TAPIR, which processes requests over UDP using the POSIX API, and kernel-bypass TAPIR, which uses the Demikernel's DPDK library OS. We compare each version to the original, non-persistent application.

We evaluate TAPIR performance using the Retwis benchmark [113], a Twitter-like transactional workload, with 10 million keys (where keys and values are 64 B) and a Zipf coefficient of 0.75. On the server side, we configure one shard with three replicas running on separate machines equipped with PM. For clients, we use a multi-process closed-loop load generator that processes RPCs over UDP using the POSIX API; it supports both the regular TAPIR and Demikernel protocols.

Table 5 reports the mean latency and peak throughput for Retwis transactions. Persimmon incurs negligible latency overhead for both the Linux and the kernel-bypass setups. For peak transaction throughput, Persimmon incurs a 5.4% degradation on Linux and a 7.3% degradation for kernel bypass. Note that the TAPIR transaction latency is much higher than the Redis latency from § 7.2.1 because each transaction involves multiple RPCs sent to three replicas; and kernel bypass provides less benefit for TAPIR than for Redis because TAPIR performs more work per RPC and its code base is less heavily optimized. Figure 6 shows the full throughput vs latency plot for this workload (lower and to the right is better). After a crash, Persimmon can recover a replica within 7 s; we were not able to compare to a recovery baseline as TAPIR has not implemented recovery from another replica.

[6] We excluded the optimization that skips newly allocated regions because no new regions are allocated during these experiments (since our workload only overwrites existing keys). However, this optimization helps tremendously when loading the initial YCSB database.

Table 5: Summary of TAPIR performance on Retwis (Zipf = 0.75, three replicas). The persistence options are volatile and Persimmon ("PSM"), with our performance in bold. Shown are the mean transaction latency at low load (measured using five clients) and peak throughput (with at most 20 clients). Persimmon incurs negligible latency overhead and a 5%–8% throughput overhead. Figure 6 shows the full latency vs throughput graph.

	Latency (µs)		**Throughput** (txn/s)	
TAPIR Server	Volatile	**PSM**	Volatile	**PSM**
Linux	342	**338**	37 K	**35 K**
Bypass	310	**309**	41 K	**38 K**

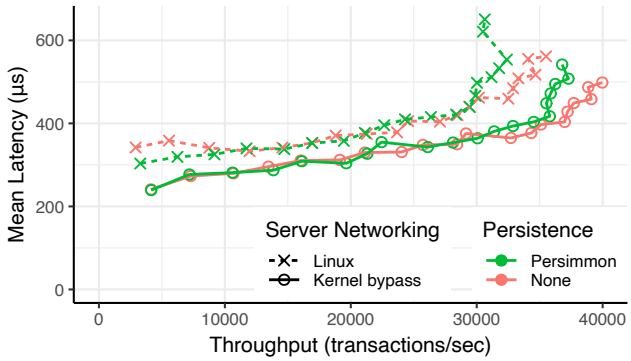
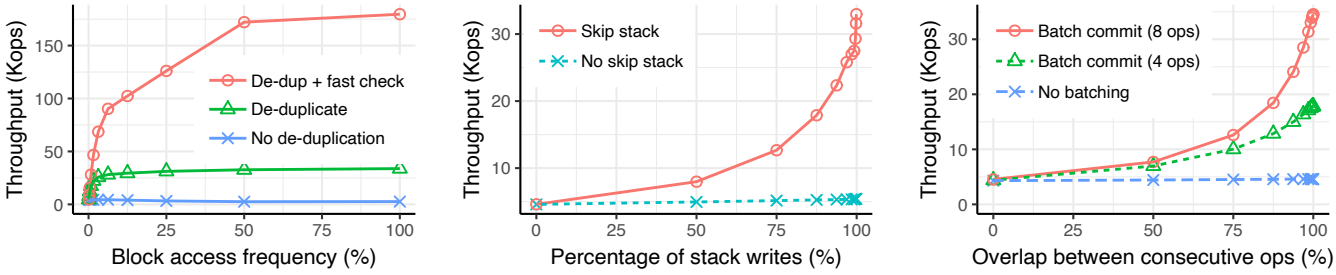


Figure 6: Latency vs throughput for TAPIR on the Retwis benchmark (10 million keys, Zipf = 0.75, median of five runs). Throughput starts decreasing as more clients are added due to congestion collapse.

7.2.3 Optimization Microbenchmarks

We use microbenchmarks to demonstrate the effectiveness of Persimmon's optimizations for crash-consistent shadow execution (§ 4.3). Each microbenchmark repeatedly invokes operations using `psm_invoke_rw`. Each operation performs 1024 memory accesses, where each access reads a 32 B *block* of memory into a `%ymm` register, performs an AVX-2 vector addition on it, and writes it back to the same memory location. We picked the access size of 32 B to match the undo logging granularity (§ 4.2). For clarity, we turn off batch commit unless otherwise specified. In each benchmark, we disable certain optimization(s) and use a memory access location pattern that demonstrates the effect of the optimization(s).

In Figure 7a, we disable undo log de-duplication and its fast-path check and vary the access frequency of each block—ranging from each operation accessing 1024 different



(a) *Undo log de-duplication* is most effective when few locations are accessed repeatedly.

(b) *Skipping stack operations* is most effective with a large percentage of stack accesses.

(c) *Batch commit* is most effective when operations access a lot of overlapping memory.

Figure 7: Effectiveness of Persimmon's optimizations (median of five runs).

blocks sequentially to accessing a single block 1024 times. When memory accesses are concentrated on few locations, de-duplication can deliver up to 13× throughput increase, and the fast-path check, an additional 5×. These optimizations incur no discernible overhead when there is no duplication.

In Figure 7b, we disable the optimization that skips stack accesses and direct a percentage of memory accesses to an array allocated on the stack. As expected, this optimization is most effective when state machine operations frequently access the stack, which we assume to be not persistent.

In Figure 7c, we enable the batch commit optimization with fixed batch sizes of 4 or 8 operations, and vary the percentage of overlap between blocks accessed by consecutive operations. Batch commit is most effective when it can group together many operations that access common memory locations.

8 Related Work

PM frameworks. Because PM's low level interface (load, store, flush) can be hard to use, prior works have proposed PM frameworks that provide higher-level APIs [9,16,18,22,31–33, 41,42,45,60,65,67,81,88,95,99,102,114]. Such a framework typically requires the programmer to (1) explicitly declare persistent data (e.g., by using a special `malloc`), (2) delineate failure-atomic regions using begin/end annotations, and (3) annotate operations that modify persistent data. The framework can then provide durability and failure atomicity by executing extra logic for each persistent operation, e.g., to log the operation and flush any modified persistent locations. Some of these frameworks reduce the programming burden by, e.g., inferring failure-atomic regions from existing synchronization points [9, 33, 41, 45, 60, 102], and by automatically interposing on persistent operations using language features [22, 95], static compiler instrumentation [9, 32, 33, 42, 102], or runtime methods [41, 88, 102].

Using these frameworks comes with two difficulties. First, despite their high-level APIs, porting an application using these frameworks can still be labor-intensive [66, 104] and bug-prone [61, 62]; we elaborate on the porting experience later in this section. Second, PM accesses and failure atomicity mechanisms on the critical path can impose significant performance overhead, especially for frameworks that implement automatic interposition. For example, microbenchmarks by Hsu et al. [41, § 5.6] show a 5×–25× slowdown for NVthreads and a 70×–200× slowdown for Mnemosyne and Atlas on memory accesses. While such overhead might be acceptable for I/O-bound applications, it can significantly slow down high-performance data stores with fast I/O.

Persimmon alleviates both difficulties. On the programming effort side, Persimmon does not require manually annotating of every persistent allocation; it simply persists all state encapsulated by the state machine. We are not aware of any other framework of its kind that provides this feature.[7] Furthermore, by taking a full-process approach to persisting the shadow process machine, Persimmon rules out referential integrity bugs [16] (e.g., arising from PM-to-DRAM pointers) and relocation issues (where a persistent region is mapped to a different address after recovery).

For performance, Persimmon keeps the critical path execution as close to the original application as possible. As far as we know, Persimmon is the only system that can transparently retain the application's memory allocator (rather than swap in a special PM allocator), thus preserving its memory layout.[8] It also pushes all PM accesses (besides operation logging) and instrumentation into the background; this requires an extra CPU core, but minimizes overhead on the critical path.

Finally, Persimmon is the first system to use *dynamic binary instrumentation* to provide failure atomicity on PM. This allows application code to call into libraries that are dynamically linked or whose source is not available (§ 4).

Porting data structures to PM. Despite the high-level APIs provided by the PM frameworks, porting existing

[7] Except AutoPersist [88], a Java PM framework that only requires marking a "durable root" object from which all persistent state can be reached. However, it exploits properties of Java / the JVM and cannot be easily extended to support C/C++ applications.

[8] Romulus [22] can be used with any memory allocator but requires manual modification to the allocator code.

data structures to PM remains challenging.[9] For example, Marathe et al. called their experience porting `memcached` "surprisingly non-trivial" [66], and Xu et al. reported five difficulties in porting Redis' hash table using PMDK, e.g., supporting Redis' many object encodings and having to order persistent writes carefully [104, §3.3]. With the high manual effort, bugs are likely to appear in PM code even when using high-level PM frameworks [61, 62]. In contrast, Persimmon requires minimal code changes to port an application (§ 7.1); in particular, we encountered none of the five difficulties identified by Xu et al. as we ported Redis using Persimmon.

Other works have noted that certain classes of data structures are easier to convert to PM and proposed techniques accordingly. For example, RECIPE [57] observes that concurrent indexes that implement helping and non-blocking reads are "inherently crash-consistent"; MOD [39] notes that purely functional data structures can be easily made failure-atomic through copy-on-write; and Friedman et al. [30] automatically transform a special class of lock-free data structures to be persistent by exploiting the *traversal* phase in these data structures' operations. Persimmon makes no such assumptions on the application's data structures.

Like Persimmon, Pronto [68] relies on state machine-like assumptions on the data structure—that it is encapsulated and has deterministic operations. This enables Pronto to implement persistence using semantic logging and periodic snapshotting. In contrast, Persimmon allows porting an entire application, rather than a single data structure, and maintains low latency even for large data sizes as it avoids the periodic stalls from the synchronous phase of snapshotting.

PM data structures / stores. Many prior works have redesigned in-memory data structures to be durable in PM. These include both tree-based [1, 10, 11, 14, 17, 44, 53, 56, 93, 97, 109] and hash-based structures [13, 25, 74, 87, 116–118]. There have also been hybrid designs that combine PM with DRAM [12,43,77,100,103,110] and/or with SSD [49,51,110]. To achieve high performance, these data structures often use data layouts and operations specifically optimized for PM.

In contrast, Persimmon is not one data structure/store; rather, it allows porting an in-memory storage system to be persistent on PM. Although a Persimmon-transformed system might not achieve resource utilization on par with hand-crafted PM data stores, Persimmon is more general and can deliver good performance on real-life workloads.

Persimmon's design is similar to that of Bullet [43], a persistent key-value store that serves requests from a "front-end cache" in DRAM, records operations in persistent logs, and uses background threads to apply logged operations to a persistent hash table. While Bullet only supports a limited set of operations, Persimmon generalizes the design to support general application-level operations. A future direction is to incorporate Bullet's cross-referencing logs into Persimmon to better support multi-core applications.

[9] Here we focus on the *porting* experience and thus do not include works that *replace* an application's data structure with a persistent one (e.g., swapping out the hash table of Redis with a persistent B-tree [93]).

PM-aware file systems. A natural way of using PM is to view it as a fast storage device and incorporate it into the storage stack. Many file systems have been designed to effectively exploit the high performance of PM [15, 20, 27–29, 48, 50, 55, 64, 94, 101, 105–107, 115]. These PM-aware file systems can transparently speed up durable applications that perform I/O using the file system interface, but do not directly apply to in-memory applications that do not use the file system.

Logging for crash consistency. Logging has been used to implement crash consistency in many contexts outside of PM, e.g., in database management systems [4, 5, 34–37, 54, 70] and journaling file systems [6, 38, 89–91]. Persimmon's logging mechanisms are conceptually similar, with a key difference being that we perform undo logging on application-supplied state machine operations (in x86-64) as opposed to SQL transactions or file system operations.

9 Conclusion

Persistent memory (PM) offers a promising solution to providing crash recovery to in-memory storage systems. However, manually porting applications to PM remains challenging. Persimmon leverages PM to provide persistence to existing in-memory storage systems while maintaining high performance and requiring minimal code changes. We use Persimmon to add persistence to Redis and TAPIR with ease while incurring minimal performance overhead on common workloads.

Acknowledgments

We thank our shepherd Vijay Chidambaram and the anonymous reviewers for their helpful comments on the paper. We'd also like to thank Aurojit Panda, David Culler, Vivian Fang, Amy Ousterhout, and other members of the UC Berkeley NetSys Lab, the RISELab, and the Microsoft Systems Research Group for their feedback. This work was funded in part by NSF Grants 1817115, 1817116, and 1704941, and by grants from Intel, VMware, Ericsson, Futurewei, and Cisco.

References

[1] ARULRAJ, J., LEVANDOSKI, J., MINHAS, U. F., AND LARSON, P.-A. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow. 11*, 5 (Jan. 2018), 553–565.

[2] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), ACM, pp. 53–64.

[3] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. of ASPLOS* (2000).

[4] BERNSTEIN, P. A., GOODMAN, N., AND HADZILACOS, V. Recovery algorithms for database systems. In *Proceedings of the IFIP 9th World Computer Congress* (1983).

[5] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency control and recovery in database systems*, vol. 370. Addison-wesley Reading, 1987.

[6] BEST, S. JFS log: How the journaled file system performs logging. In *Annual Linux Showcase & Conference* (2000).

[7] BORNHOLT, J., KAUFMANN, A., LI, J., KRISHNAMURTHY, A., TORLAK, E., AND WANG, X. Specifying and checking file system crash-consistency models. In *Proc. of ASPLOS* (2016), pp. 83–98.

[8] BRUENING, D., ZHAO, Q., AND AMARASINGHE, S. Transparent dynamic instrumentation. In *Proc. of VEE* (2012).

[9] CHAKRABARTI, D. R., BOEHM, H.-J., AND BHANDARI, K. Atlas: Leveraging locks for non-volatile memory consistency. In *Proc. of OOPSLA* (2014).

[10] CHEN, S., GIBBONS, P. B., AND NATH, S. Rethinking database algorithms for phase change memory. In *Proc. of CIDR* (2011).

[11] CHEN, S., AND JIN, Q. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow. 8*, 7 (Feb. 2015), 786–797.

[12] CHEN, Y., LU, Y., YANG, F., WANG, Q., WANG, Y., AND SHU, J. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proc. of ASPLOS* (2020).

[13] CHEN, Z., HUANG, Y., DING, B., AND ZUO, P. Lock-free concurrent level hashing for persistent memory. In *Proc. of USENIX ATC* (2020).

[14] CHI, P., LEE, W.-C., AND XIE, Y. Making B+-tree efficient in PCM-based main memory. In *Proc. of ISLPED* (2014).

[15] CHOI, J., HONG, J., KWON, Y., AND HAN, H. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *Proc. of USENIX ATC* (2020).

[16] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of ASPLOS* (2011).

[17] COHEN, N., AKSUN, D. T., AVNI, H., AND LARUS, J. R. Fine-grain checkpointing with in-cache-line logging. In *Proc. of ASPLOS* (2019).

[18] COHEN, N., AKSUN, D. T., AND LARUS, J. R. Object-oriented recovery for non-volatile memory. *Proc. ACM Program. Lang. 2*, OOPSLA (Oct. 2018).

[19] COHEN, N., FRIEDMAN, M., AND LARUS, J. R. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017).

[20] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proc. of SOSP* (2009).

[21] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proc. of SOCC* (2010).

[22] CORREIA, A., FELBER, P., AND RAMALHETE, P. Romulus: Efficient algorithms for persistent transactional memory. In *Proc. of SPAA* (2018).

[23] cpython: 52f68c95e025 objects/dictobject.c. https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c#l33.

[24] CRIU. https://criu.org.

[25] DEBNATH, B., HAGHDOOST, A., KADAV, A., KHATIB, M. G., AND UNGUREANU, C. Revisiting hash table design for phase change memory. In *Proc. of INFLOW* (2015).

[26] demikernel/demikernel: Demikernel OS. https://github.com/demikernel/demikernel.

[27] DONG, M., BU, H., YI, J., DONG, B., AND CHEN, H. Performance and protection in the ZoFS user-space NVM file system. In *Proc. of SOSP* (2019).

[28] DONG, M., AND CHEN, H. Soft updates made simple and fast on non-volatile memory. In *Proc. of USENIX ATC* (2017).

[29] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proc. of EuroSys* (2014).

[30] FRIEDMAN, M., BEN-DAVID, N., WEI, Y., BLELLOCH, G. E., AND PETRANK, E. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *Proc. of PLDI* (2020).

[31] GENÇ, K., BOND, M. D., AND XU, G. H. Crafty: Efficient, HTM-compatible persistent transactions. In *Proc. of PLDI* (2020).

[32] GEORGE, J. S., VERMA, M., VENKATASUBRAMANIAN, R., AND SUBRAHMANYAM, P. go-pmem: Native support for programming persistent memory in Go. In *Proc. of USENIX ATC* (2020).

[33] GOGTE, V., DIESTELHORST, S., WANG, W., NARAYANASAMY, S., CHEN, P. M., AND WENISCH, T. F. Persistency for synchronization-free regions. In *Proc. of PLDI* (2018).

[34] GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the System R database manager. *ACM Computing Surveys (CSUR)* (1981).

[35] GRAY, J., AND REUTER, A. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[36] GRAY, J. N. Notes on data base operating systems. In *Operating Systems*. Springer, 1978.

[37] HAERDER, T., AND REUTER, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (1983).

[38] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proc. of SOSP* (1987).

[39] HARIA, S., HILL, M. D., AND SWIFT, M. M. MOD: Minimally ordered durable datastructures for persistent memory. In *Proc. of ASPLOS* (2020).

[40] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[41] HSU, T. C.-H., BRÜGNER, H., ROY, I., KEETON, K., AND EUGSTER, P. NVthreads: Practical persistence for multi-threaded applications. In *Proc. of EuroSys* (2017).

[42] HU, Q., REN, J., BADAM, A., SHU, J., AND MOSCIBRODA, T. Log-structured non-volatile main memory. In *Proc. of USENIX ATC* (2017).

[43] HUANG, Y., PAVLOVIC, M., MARATHE, V. J., SELTZER, M., HARRIS, T., AND BYAN, S. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *Proc. of USENIX ATC* (2018).

[44] HWANG, D., KIM, W.-H., WON, Y., AND NAM, B. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proc. of FAST* (2018).

[45] IZRAELEVITZ, J., KELLY, T., AND KOLLI, A. Failure-atomic persistent memory updates via JUSTDO logging. In *Proc. of ASPLOS* (2016).

[46] IZRAELEVITZ, J., YANG, J., ZHANG, L., KIM, J., LIU, X., MEMARIPOUR, A., SOH, Y. J., WANG, Z., XU, Y., DULLOOR, S. R., ZHAO, J., AND SWANSON, S. Basic performance measurements of the Intel Optane DC Persistent Memory Module. *CoRR abs/1903.05714* (2019). http://arxiv.org/abs/1903.05714.

[47] jemalloc. http://jemalloc.net/.

[48] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proc. of SOSP* (2019).

[49] KAIYRAKHMET, O., LEE, S., NAM, B., NOH, S. H., AND CHOI, Y.-R. SLM-DB: Single-level key-value store with persistent memory. In *Proc. of FAST* (2019).

[50] KANNAN, S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., WANG, Y., XU, J., AND PALANI, G. Designing a true direct-access file system with DevFS. In *Proc. of FAST* (2018).

[51] KANNAN, S., BHAT, N., GAVRILOVSKA, A., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proc. of USENIX ATC* (2018).

[52] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about Eve: Execute-verify replication for multi-core servers. In *Proc. of OSDI* (2012).

[53] KIM, W.-H., SEO, J., KIM, J., AND NAM, B. ClfB-Tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Trans. Storage 14*, 1 (Feb. 2018).

[54] KUMAR, V., AND HSU, M. *Recovery mechanisms in database systems*. Prentice Hall PTR, 1997.

[55] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proc. of SOSP* (2017).

[56] LEE, S. K., LIM, K. H., SONG, H., NAM, B., AND NOH, S. H. WORT: Write optimal radix tree for persistent memory storage systems. In *Proc. of FAST* (2017).

[57] LEE, S. K., MOHAN, J., KASHYAP, S., KIM, T., AND CHIDAMBARAM, V. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proc. of SOSP* (2019).

[58] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proc. of OSDI* (2016).

[59] LINUX KERNEL ORGANIZATION. Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[60] LIU, Q., IZRAELEVITZ, J., LEE, S. K., SCOTT, M. L., NOH, S. H., AND JUNG, C. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *Proc. of MICRO* (2018).

[61] LIU, S., SEEMAKHUPT, K., WEI, Y., WENISCH, T., KOLLI, A., AND KHAN, S. Cross-failure bug detection in persistent memory programs. In *Proc. of ASPLOS* (2020).

[62] LIU, S., WEI, Y., ZHAO, J., KOLLI, A., AND KHAN, S. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proc. of ASPLOS* (2019).

[63] LU, H. J., MATZ, M., GIRKAR, M., HUBIČKA, J., JAEGER, A., AND MITCHELL, M. System V application binary interface AMD64 architecture processor supplement (with LP64 and ILP32 programming models) version 1.0, 2018. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf.

[64] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: An RDMA-enabled distributed persistent memory file system. In *Proc. of USENIX ATC* (2017).

[65] MARATHE, V. J., MISHRA, A., TRIVEDI, A., HUANG, Y., ZAGHLOUL, F., KASHYAP, S., SELTZER, M., HARRIS, T., BYAN, S., BRIDGE, B., AND DICE, D. Persistent memory transactions. *CoRR abs/1804.00701* (2018). http://arxiv.org/abs/1804.00701.

[66] MARATHE, V. J., SELTZER, M., BYAN, S., AND HARRIS, T. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *Proc. of HotStorage* (2017).

[67] MEMARIPOUR, A., BADAM, A., PHANISHAYEE, A., ZHOU, Y., ALAGAPPAN, R., STRAUSS, K., AND SWANSON, S. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *Proc. of EuroSys* (2017).

[68] MEMARIPOUR, A., IZRAELEVITZ, J., AND SWANSON, S. Pronto: Easy and fast persistence for volatile data structures. In *Proc. of ASPLOS* (2020).

[69] MICHAEL, E., PORTS, D. R., SHARMA, N. K., AND SZEKERES, A. Recovering shared objects without stable storage. In *Proc. of DISC* (2017).

[70] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* (1992).

[71] MOHAN, J., MARTINEZ, A., PONNAPALLI, S., RAJU, P., AND CHIDAMBARAM, V. Finding crash-consistency bugs with bounded black-box crash testing. In *Proc. of OSDI* (2018).

[72] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proc. of SOSP* (2013).

[73] NALLI, S., HARIA, S., HILL, M. D., SWIFT, M. M., VOLOS, H., AND KEETON, K. An analysis of persistent memory use with WHISPER. In *Proc. of ASPLOS* (2017).

[74] NAM, M., CHA, H., CHOI, Y.-R., NOH, S. H., AND NAM, B. Write-optimized dynamic hashing for persistent memory. In *Proc. of FAST* (2019).

[75] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., ET AL. Scaling memcache at Facebook. In *Proc. of NSDI* (2013).

[76] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proc. of SOSP* (2011).

[77] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proc. of SIGMOD* (2016).

[78] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proc. of NSDI* (2019).

[79] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proc. of OSDI* (2014).

[80] Persistent memory development kit. https://pmem.io/pmdk/.

[81] Persistent memory development kit—the libpmemobj library. https://pmem.io/pmdk/libpmemobj/.

[82] pmem/pmem-redis: A version of redis that uses persistent memory. https://github.com/pmem/pmem-redis.

[83] Protocol Buffers | Google Developers. https://developers.google.com/protocol-buffers/.

[84] RAAD, A., WICKERSON, J., NEIGER, G., AND VAFEIADIS, V. Persistency semantics of the Intel-X86 architecture. *Proc. ACM Program. Lang. 4*, POPL (Dec. 2019).

[85] Redis persistence. https://redis.io/topics/persistence.

[86] redis/README.md at 4.0 • redis/redis. https://github.com/redis/redis/blob/4.0/README.md.

[87] SCHWALB, D., DRESELER, M., UFLACKER, M., AND PLATTNER, H. NVC-Hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proc. of IMDM* (2015).

[88] SHULL, T., HUANG, J., AND TORRELLAS, J. AutoPersist: An easy-to-use Java NVM framework based on reachability. In *Proc. of PLDI* (2019).

[89] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proc. of ATEC* (1996).

[90] TWEEDIE, S. EXT3, journaling filesystem, 2000. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html.

[91] TWEEDIE, S. C., ET AL. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo* (1998).

[92] VAN RENEN, A., VOGEL, L., LEIS, V., NEUMANN, T., AND KEMPER, A. Persistent memory I/O primitives. In *Proc. of DaMoN* (2019).

[93] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of FAST* (2011).

[94] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. of EuroSys* (2014).

[95] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proc. of ASPLOS* (2011).

[96] WANG, C., YING, V., AND WU, Y. Supporting legacy binary code in a software transaction compiler with dynamic binary translation and optimization. In *Proc. of CC* (2008).

[97] WANG, T., LEVANDOSKI, J., AND LARSON, P. Easy lock-free indexing in non-volatile memory. In *Proc. of ICDE* (2018).

[98] swapnilh/whisper: WHISPER is a comprehensive benchmark suite for emerging persistent memory technologies. https://github.com/swapnilh/whisper.

[99] WU, M., ZHAO, Z., LI, H., LI, H., CHEN, H., ZANG, B., AND GUAN, H. Espresso: Brewing Java for more non-volatility with non-volatile memory. In *Proc. of ASPLOS* (2018).

[100] WU, X., NI, F., ZHANG, L., WANG, Y., REN, Y., HACK, M., SHAO, Z., AND JIANG, S. NVMcached: An NVM-based key-value cache. In *Proc. of APSys* (2016).

[101] WU, X., QIU, S., AND NARASIMHA REDDY, A. L. SCMFS: A file system for storage class memory and its extensions. *ACM Trans. Storage 9*, 3 (Aug. 2013).

[102] WU, Z., LU, K., NISBET, A., ZHANG, W., AND LUJÁN, M. PMThreads: Persistent memory threads harnessing versioned shadow copies. In *Proc. of PLDI* (2020).

[103] XIA, F., JIANG, D., XIONG, J., AND SUN, N. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proc. of USENIX ATC* (2017).

[104] XU, J., KIM, J., MEMARIPOUR, A., AND SWANSON, S. Finding and fixing performance pathologies in persistent memory software stacks. In *Proc. of ASPLOS* (2019).

[105] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. of FAST* (2016).

[106] XU, J., ZHANG, L., MEMARIPOUR, A., GANGADHARAIAH, A., BORASE, A., DA SILVA, T. B., SWANSON, S., AND RUDOFF, A. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proc. of SOSP* (2017).

[107] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. Orion: A distributed file system for non-volatile main memories and RDMA-capable networks. In *Proc. of FAST* (2019).

[108] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *Proc. of FAST* (2020).

[109] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proc. of FAST* (2015).

[110] YAO, T., ZHANG, Y., WAN, J., CUI, Q., TANG, L., JIANG, H., XIE, C., AND HE, X. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *Proc. of USENIX ATC* (2020).

[111] brianfrankcooper/ycsb: Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[112] ZHANG, I., LIU, J., AUSTIN, A., ROBERTS, M. L., AND BADAM, A. I'm not dead yet! The role of the operating system in a kernel-bypass era. In *Proc. of HotOS* (2019).

[113] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURHTY, A., AND PORTS, D. R. K. Building consistent transactions with inconsistent replication. In *Proc. of SOSP* (2015).

[114] ZHANG, L., AND SWANSON, S. Pangolin: A fault-tolerant persistent memory programming library. In *Proc. of USENIX ATC* (2019).

[115] ZHENG, S., HOSEINZADEH, M., AND SWANSON, S. Ziggurat: A tiered file system for non-volatile main memories and disks. In *Proc. of FAST* (2019).

[116] ZUO, P., AND HUA, Y. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems 29*, 5 (2018).

[117] ZUO, P., HUA, Y., AND WU, J. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proc. of OSDI* (2018).

[118] ZURIEL, Y., FRIEDMAN, M., SHEFFI, G., COHEN, N., AND PETRANK, E. Efficient lock-free durable sets. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct. 2019).



AGAMOTTO: How Persistent is your Persistent Memory Application?

Ian Neal
University of Michigan

Ben Reeves
University of Michigan

Ben Stoler
University of Michigan

Andi Quinn *University of Michigan*

Youngjin Kwon
KAIST

Simon Peter
University of Texas at Austin

Baris Kasikci
University of Michigan

Abstract

Persistent Memory (PM) can be used by applications to directly and quickly persist any data structure, without the overhead of a file system. However, writing PM applications that are simultaneously correct and efficient is challenging. As a result, PM applications contain correctness and performance bugs. Prior work on testing PM systems has low bug coverage as it relies primarily on extensive test cases and developer annotations.

In this paper we aim to build a system for more thoroughly testing PM applications. We inform our design using a detailed study of 63 bugs from popular PM projects. We identify two application-independent patterns of PM misuse which account for the majority of bugs in our study and can be detected automatically. The remaining application-specific bugs can be detected using compact custom oracles provided by developers.

We then present AGAMOTTO, a generic and extensible system for discovering misuse of persistent memory in PM applications. Unlike existing tools that rely on extensive test cases or annotations, AGAMOTTO symbolically executes PM systems to discover bugs. AGAMOTTO introduces a new symbolic memory model that is able to represent whether or not PM state has been made persistent. AGAMOTTO uses a state space exploration algorithm, which drives symbolic execution towards program locations that are susceptible to persistency bugs. AGAMOTTO has so far identified 84 new bugs in 5 different PM applications and frameworks while incurring no false positives.

1 Introduction

Persistent Memory (PM) is a promising new technology that offers an appealing performance-cost tradeoff for application developers. PM technologies, such as Intel Optane DC [36], can offer persistent memory accesses with latencies that are only 2–3× higher than the latencies of DRAM [70]. Moreover, such PM technologies are cheaper than DRAM per GB of capacity [3]. As byte-addressable memory, PM can also be accessed via processor load and store instructions. Application developers have already started building systems that use PM directly, without relying on heavyweight system calls to ensure durability, including ports of popular systems such as memcached [24] and Redis [21].

While using PM directly via persistent data structures can offer performance, it is challenging to write PM-based applications that are simultaneously correct and efficient [12, 18, 33, 52, 54, 60, 71, 76]. Persistent memory writes in the CPU cache must be explicitly flushed to PM using specific instructions or APIs. In certain cases, PM flush operations need to be ordered using memory fences to enforce crash consistency. Incorrect usage of these mechanisms can result in *persistency bugs* which break crash-consistency guarantees or degrade application performance. Persistency bugs are challenging to diagnose because their symptoms are easily masked. For example, crash-consistency bugs may be masked because PM writes are implicitly flushed when dirty (or updated) cache lines are evicted from the CPU—furthermore, flushes which are required for proper crash consistency under one execution path may be redundant and unnecessary under a different program execution path, leading to performance degradations.

Several systems have been built to aid with testing PM applications; however, these existing approaches are either specific to a target application or require significant manual developer effort. Intel designed Yat [44] and pmemcheck [65] specifically to test the crash consistency and durability of PMFS (Persistent Memory File System) [27] and PMDK (Persistent Memory Development Kit) [20], respectively. To find bugs, Yat exhaustively tests all possible update orderings, and pmemcheck tracks annotated updates. Both of these tools are specific to a single system (PMFS and PMDK, respectively) and are hard to generalize. Other tools like Persistency Inspector [62], PMTest [50], and XFDetector [49] are applicable to general PM systems, but require developer annotations and extensive test suites to thoroughly test PM applications.

In order to determine the extent to which persistency bug finding can be automated (i.e., not require program annota-

tions) to test general systems, we perform a study of 63 bugs in PM applications and frameworks. We identify two application-independent patterns of PM misuse (*missing flush/fence* and *extra flush/fence*) which cover the majority (89%, or 56 out of 63) of bugs in our study and can be detected automatically. The remaining bugs are application-specific; for example, many of the remaining bugs involve misusing transactions when updating data-structures. Existing PM testing approaches do not identify application-independent patterns of misuse, and therefore require annotations to detect any PM bug. In addition to classifying bugs based on their pattern of PM misuse, we also classify bugs based on whether they affect performance or correctness.

Based on the insights gained through our study, we present AGAMOTTO, a framework for detecting bugs in PM applications that does not rely on extensive test cases. Instead, AGAMOTTO uses symbolic execution [8] to thoroughly explore the state space of a program. In addition to expanding path coverage, symbolic execution also allows AGAMOTTO to detect persistency bugs in an application without access to underlying physical PM resources. AGAMOTTO introduces a memory model to track updates made to PM by the explored program paths, and supports *bug oracles* which use the PM state to identify bugs in the program. AGAMOTTO automatically detects persistency bugs using two *universal persistency bug oracles* based on the common patterns of PM misuse identified by our study. The first is an *unflushed/unfenced* oracle that identifies modifications to PM cache lines that are not flushed or fenced (both a correctness and performance issue) and the second an *extra-flushed/fenced* oracle that identifies duplicate flushes of the same cache line or unnecessary fences (a performance issue [18, 52, 60, 71, 76]).

To identify application-specific persistency bugs, AGAMOTTO allows developers to provide *custom persistency bug oracles*. To demonstrate the versatility of custom oracles, we implemented two such oracles in AGAMOTTO to detect bugs related to misuse of the PMDK transactional API [20, 49, 50].

Analyzing large PM applications using traditional symbolic execution [8] leads to scalability issues since the state space of possible executions grows exponentially with the size of the analyzed program. AGAMOTTO uses a novel search algorithm that prunes the execution states it analyzes, allowing AGAMOTTO to discover more bugs. Prior to symbolic execution, AGAMOTTO uses a whole-program static analysis to determine instructions that modify PM (stores, flushes, etc.) and assigns a unit priority to them. AGAMOTTO then assigns an aggregate priority to each instruction by back-propagating the unit priorities from each PM-modifying instruction—this makes the aggregate priority a measure of the number of PM-modifying instructions reachable from a particular instruction. AGAMOTTO uses priorities to steer symbolic execution into program states that frequently modify PM.

We used AGAMOTTO to find 84 new persistency bugs in real-world systems including PMDK (a mature PM library) [20], memcached-pm [24], Redis-pmem [21], NVM-Direct [7], and RECIPE [45]. In particular, we found 13 new correctness and 70 new performance bugs using the universal persistency bug oracles, and 1 new correctness bug using a custom persistency bug oracle. We report all bugs to their authors, and so far 40 of them have been confirmed and none denied.

In this paper we make the following contributions:

- We perform a detailed study of persistency bugs in PMDK as well as bugs found by prior work, and present a new taxonomy of persistency bugs.
- We build AGAMOTTO[1], a persistency bug detection tool that can test real-world PM programs using a novel state exploration algorithm. AGAMOTTO automatically detects bugs using two universal persistency bug oracles, without relying on user annotations or an extensive test suite. AGAMOTTO is extensible with custom bug oracles that can detect application-specific bugs.
- We use AGAMOTTO to find 84 new bugs in 5 applications and persistent memory libraries, compared to the 6 persistency bugs found in persistent applications by the state of the art (PMTest [50], which finds 3 bugs, and XFDetector [49], which finds 3 bugs). AGAMOTTO does not incur any false positives in our evaluation.

In the rest of this paper, we first provide background on PM programming and describe the challenges of PM bug finding (§2). We then present the results of our PM bug study and provide common patterns of PM misuse that identify PM bugs (§3). Then, we discuss the persistency bug detection algorithms and search techniques underlying AGAMOTTO (§4). Next, we describe the high-level design of AGAMOTTO and evaluate the system with respect to both the number of bugs found and the impact of these bugs (§6). Finally, we describe related PM bug detection work (§7).

2 Background and Challenges

We now provide a background on persistent memory (PM) programming and difficulties associated with writing correct and efficient PM programs.

2.1 Persistent Memory Programming

```
1 int *x = pm_alloc(), *y = pm_alloc();
2 *x = 1;
3 clwb(x)
4 sfence()
5 *y = 1;
6 clwb(y)
7 sfence()
```

Listing 1: A PM programming example.

PM implementations support a programming interface that diverges from that of conventional storage devices. Rather than

[1] Released at https://github.com/efeslab/agamotto

using comparatively slow system calls to access persistent memory, applications can accelerate PM accesses by directly mapping pages of PM into their address space and performing byte-addressable load/store operations. Like volatile memory accesses, PM IO may be cached and buffered in volatile memory (i.e., the CPU cache) in order to increase performance.

The added performance comes at the cost of increased complexity for the application developer. Volatile memory can retain updates to PM for an indefinite period of time (e.g., until a cache line gets evicted). Ensuring that stores to PM are durable requires two steps. First, a developer must issue a *flush* for the cache-line that contains the updated data. Then, the developer orders flushes using existing fence operations (e.g., `SFENCE`). Note that an unordered flush may not be written to persistent memory before a crash, so fences are required for durability. Consider Listing 1, which allocates two integers in persistent memory and issues ordered writes to the integers. In order to guarantee that the write to `x` (line 2) is ordered before the write to `y` (line 5), a flush and fence must occur between the updates (lines 3 and 4). To ensure that the write to `y` (line 5) is durable, a flush and fence must occur after the write (lines 6 and 7).

The x86 instruction set architecture (ISA) provides two flush instructions: `CLFLUSHOPT` and `CLWB`. `CLWB` differs from `CLFLUSHOPT` in that `CLWB` hints the CPU to keep the cache line in the cache whereas `CLFLUSHOPT` does not. x86 provides two fence instructions: `MFENCE`, which orders all loads, stores, and flushes; and `SFENCE`, which orders all stores and all flushes. Additionally, x86 provides `CLFLUSH`, which acts as both a flush and fence for a specific cache line (i.e., only orders the flush that the `CLFLUSH` itself issues, other `CLWB` and `CLFLUSHOPT` instructions must be ordered by a separate fence). Finally, x86 allows non-temporal stores, which bypass the cache and thus do not require a flush but do require a fence for durability. Note that the classification of PM instructions into flush and fence operations is not x86-specific. For example, ARM provides flush (e.g., `DC CVAP`) and fence (e.g., `DSB`) operations [5, 67] with similar semantics to x86 flushes and fences.

2.2 Challenges of Detecting PM Bugs

PM interfaces for durability and performance are easy to misuse [49, 50] and the resulting persistency bugs can be challenging to detect. Persistency bugs exhibit many characteristics that make them difficult to detect. First, finding a persistency bug requires identifying whether PM cache-lines are dirty, but the x86 ISA does not provide a mechanism to determine the state of a cache-line. Thus, detecting a persistency bug requires modeling PM state and instrumenting the program for tracking state updates, which is challenging to accomplish using traditional debugging tools. Second, in the case of correctness bugs, the root cause and symptoms of a persistency bug are often loosely tied together: while the symptoms of a correctness persistency bug is only revealed after a crash, the PM misuse (i.e., the root cause) may be hundreds of thousands of instructions before the crash even occurred. Finally, persistency bugs are easily masked by other system behavior. For example, flushes which are redundant in one execution path of the program may be necessary under a slightly different execution path, while correctness persistency bugs may be masked by the CPU when evicting a dirty cache-line from its cache.

Unfortunately, developers cannot solely rely on PM frameworks (e.g., PMDK [20]) to prevent these bugs. As we show in §3, many applications use PM libraries incorrectly and even these established libraries themselves may misuse PM.

Project	Missing Flush/Fence	Extra Flush/Fence	Other	Total
PMDK	49	6	2	57
PMTest	1	1	1	3
XFDetector	-	-	3	3
Total	**50**	**6**	**7**	**63**

Table 1: The results of our bug survey.

3 PM Bug Study and Classification

In this section, we present a study of persistency bugs. We construct a corpus of 63 persistency bugs from a mature PM library, PMDK [20], and persistency bugs from PM projects (PMFS [27] and Redis-pmem [21]) that were found by state-of-the-art PM bug detection tools (PMTest [50] and XFDetector [49]). We chose PMDK, because it is a mature project with a thorough issue tracker [23] representing a large collection of existing bugs. We use this corpus to identify common patterns of PM bugs.

Table 1 shows a summary of our results[2]. Overall, we find that two application-independent PM patterns explain the vast majority (56/63 bugs) of the reported persistency bugs. We find that PM bugs can result in either correctness problems, which may lead to data corruption, or performance problems. In particular, the *missing flush/fence* pattern, in which an update to persistent memory is missing subsequent flush and/or fence operations, accounts for 50/63 bugs and can lead to either correctness or performance issues. The *extra flush/fence* pattern, in which a cache-line is redundantly flushed or a fence instruction is issued that is not needed for PM durability, accounts for 6/63 bugs and leads to performance degradation. The remaining 7 are caused by application-specific violations, most of which involve a misuse of the PMDK transaction API. Note, our study may be biased towards bugs that are detectable by existing PM bug detection tools, because PMDK

[2]We provide a link to our bug study results in the AGAMOTTO GitHub repository: https://github.com/efeslab/agamotto/blob/artifact-eval-osdi20/artifact/README.md

developers extensively use pmcheck [65] to detect bugs. In the rest of this section, we present examples of these bugs together with more detailed descriptions.

3.1 Missing Flush/Fence Pattern

```
1 //oid is a pointer to PM
2 if (if_free != 0)
3   *oid = NULL;
4   // BUG: missing flush and fence
```

Listing 2: A missing flush/fence correctness bug adapted from PMDK Issue #1103, Pull Request (PR) #3907.

The most common bug pattern in the bugs in our study is the missing flush/fence pattern, in part because PMDK developers extensively use pmemcheck [65] which identifies this pattern of PM misuse. In this bug pattern, an update to PM is not made durable because it is missing a subsequent flush and/or fence operation. An example of the pattern is shown in Listing 2. Here, a pointer to persistent memory, `oid`, is not flushed when `if_free != 0`. If the program crashed and restarted, the pointer might point to its old value, which could lead to rogue writes or malformed data reads. This bug is fixed by adding proper flush and fence operations after the modification.

In contrast, the missing flush/fence pattern is detectable without any application-specific information. In our study, instances of the missing flush/fence pattern are correctness issues, where the program is unable to recover from a crash similar to the one in Listing 2. In our evaluation (see §6), we also found instances of the missing flush/fence pattern which are performance bugs. In these instances, an application uses persistent memory to store volatile data, which hinders performance due to the higher latency of PM accesses relative to DRAM accesses. Existing studies suggest that placing volatile data in PM can decrease application performance by as much as 5% [26]. There are PM data structures that intentionally include this pattern [53] as a programming simplification. However, in the applications included in our study and evaluation, all instances of the missing flush/fence pattern are persistency bugs.

3.2 Extra Flush/Fence Pattern

The other common pattern of persistent memory misuse which we identify in our study is the extra flush/fence pattern. In this pattern, a cache-line is redundantly flushed, or a fence instruction which is not needed for PM durability is executed. An example of this is shown in Listing 3. In this example, an array located in persistent memory is resized in-place using the call to `resize_array`, new elements are initialized to 0, and new elements are flushed to persistent memory. However, when the size of the array is reduced (i.e., `new_size < size`), an underflow in line 12 causes unnecessary flushes and leads to a performance degradation [18, 60, 71, 76] (e.g., an additional flush and fence can add an average of 250ns of latency [51, 73], where the base latency of uncached PM accesses can be as low as 96ns [37]).

```
1  //array is an array of integers in PM
2  //with length = size
3
4  //resizes array in-place
5  resize_array(array, new_size);
6
7  // if size >= new_size, no copying occurs
8  for (size_t i = size; i < new_size; i++)
9      array[i] = 0;
10
11 // BUG: when new_size < size, underflow!
12 for (size_t i = 0; i < new_size - size; ++i)
13     clwb(array[i + size])
14 sfence();
```

Listing 3: An extra flush/fence performance bug adapted from PMDK issue #1117, PR #3860 .

Similar to the missing flush/fence pattern, the extra flush/fence pattern is detectable without any application-specific information. The extra flush/fence pattern results in performance degradation. As flush and fence instructions are used in non-PM contexts (e.g., fences provide semantics for memory consistency), there may be instances of this pattern that are not persitency bugs. However, in the applications in our study and evaluation, all instances of the extra flush/fence pattern are persistency bugs.

3.3 Other Bugs

```
1 // store pool's header
2 /* BUG: header made valid before
3   pool data made valid */
4 header = ...
5 clwb(header);
6 sfence();
7 pool = ...
8 clwb(pool);
9 sfence();
```

Listing 4: An example correctness bug adapted from PMDK Issue #14.

The remaining 7 bugs in the study are application-specific; i.e., in these cases, data is correctly flushed to PM and there are no redundant flush operations, but the application misuses PM, leading to performance or correctness issues. For example, Listing 4 depicts a bug adapted from the memory pool allocator in PMDK which results in a correctness issue. In order to recover from a crash, the values in `header` and `pool` must be consistent; however a crash at Line 7 will result in an updated value of `header` without an updated value of `pool`.

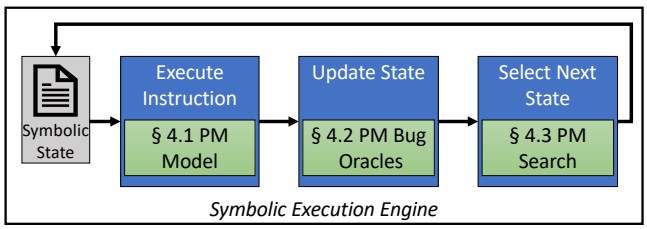


Figure 1: Components of AGAMOTTO. Green-shaded boxes are AGAMOTTO-specific.

3.4 Summary and Insights

We summarize several key results we obtained and the insights we gathered from this bug study which inform AGAMOTTO's design decisions.

- The missing flush/fence and extra flush patterns are prevalent (56/63 of the bugs we found) and application-independent. Hence, an automated approach (i.e., requiring little to no developer effort or source modification) could and should be used to detect them across a variety of platforms.
- In our study, all instances of the missing flush/fence and extra flush/fence patterns are persistency bugs; we hypothesize that this trend will hold for general PM applications. In §6, we find that all instances of these patterns are persistency bugs across a variety of PM libraries and applications.
- The remaining bugs, while less prevalent in our survey, are still potential sources of inconsistency and/or performance loss. An ideal tool should allow developers to specify application-specific patterns without requiring extensive test cases and significant developer annotations.

4 Design

In this section, we describe the design of AGAMOTTO. AGAMOTTO aims to achieve four high-level design principles:

Automation. Bug-finding can take a substantial amount of developer effort [56,68]; AGAMOTTO aims to automate as much as possible to reduce this burden. For example, AGAMOTTO is non-intrusive (i.e., requires no source-code modifications) and leverages basic test cases (e.g., existing unit tests or example code) to explore execution paths in an application.

Generality. AGAMOTTO can test any PM application.

High Accuracy. AGAMOTTO aims to report no false positives (i.e., reporting a bug where there is none) while also reducing false negatives (i.e., failure to find a bug).

Extensibility. AGAMOTTO can be easily extended to find application-specific bugs.

The major components of AGAMOTTO are shown in Fig. 1 (green-shaded boxes represent the key components unique to AGAMOTTO). AGAMOTTO relies on an existing symbolic execution engine (KLEE [8] in our prototype) to explore the state space of a PM program. During this exploration, AGAMOTTO uses a custom PM model to express and track updates to persistent memory regions (i.e., writes, flushes and fences). Since AGAMOTTO tracks PM symbolically, it does not need access to PM resources in order to detect persistency bugs in a PM application. As AGAMOTTO explores the state space of the program, it checks for PM bugs using universal bug oracles, as well as any custom bug oracles that users may provide. Universal oracles check for the missing flush/fence pattern and the extra flush/fence patterns of PM misuse identified in our study. Custom oracles can check for application-specific bugs, which may be correctness bugs (e.g., ordering bugs) and/or performance bugs (e.g., redundant transaction operations) akin to prior work [49,50].

At the heart of AGAMOTTO lies its PM-aware state space exploration algorithm, which is effective in steering symbolic execution towards program locations that exercise PM. In symbolic execution, inputs are symbolic (unconstrained) values in a program's initial state. When the program reaches a branch depending on symbolic input, the current state is forked and the constraints on input are updated depending on the branch condition. As states increase by forking, symbolic execution needs to employ a state-space exploration strategy. Existing state space exploration strategies, such as maximizing code coverage, are not optimized for finding PM bugs, and thus waste resources exploring uninteresting paths.

Instead, before symbolically executing the program, AGAMOTTO uses a custom static analysis to determine instructions that can modify persistent memory. AGAMOTTO then uses a back-propagation algorithm to assign a weight to each instruction equal to the number of PM-modifying instructions that are reachable from that instruction. AGAMOTTO prioritizes exploring the program state whose currently-executed instruction has the highest such weight. We find that the number of PM-modifying paths is much smaller than the total number of execution paths in practice, allowing AGAMOTTO to thoroughly explore the set of executions that lead to persistency bugs (see §6).

When AGAMOTTO's oracles detect a bug during state space exploration, AGAMOTTO relies on its underlying symbolic execution engine to invoke a constraint solver and determine the inputs that led to the bug, thereby creating a test case that a developer can use for debugging.

In the rest of this section we provide details regarding the key components of AGAMOTTO.

4.1 PM Model and PM State Tracking

AGAMOTTO facilitates persistency bug detection by tracking the state of persistent memory objects in the program. For each PM allocation, AGAMOTTO tracks constraints on the *persistency state* of the allocated cache lines. The persistency

state of a cache-line indicates whether the cache line is *dirty* (i.e., modified), *pending* (i.e., updates to the cache-line are flushed but not ordered) or *clean* (i.e., updates to the cache-line are both flushed and ordered). As AGAMOTTO symbolically executes, it updates constraints on the persistency state of PM cache-lines to reflect the behavior of the program. AGAMOTTO uses these constraints to identify execution paths which contain persistency bugs, (i.e., when redundant flushes are issued, or updates are not properly ordered).

Identifying PM allocations In order to be application-agnostic and automated, AGAMOTTO tracks persistent memory allocations from the system level, rather than tracking high-level calls to persistent memory allocators (e.g., `pmem_alloc`) [50]. Tracking PM allocations at a system level trades off performance in favor of automation, since this approach over-approximates PM allocations. AGAMOTTO marks all opened files that match a user-specified persistent memory device regular expression (e.g., `pmem/*`) as PM files and treats memory-mappings of PM files as persistent memory objects.

Tracking Persistent Memory State. When AGAMOTTO symbolically executes an instruction that operates on a PM object, it generates constraints on the persistency state of the cache-lines that comprise the memory objects. A store instruction (e.g., x86 `MOV`) adds a constraint that the destination of the store is in the dirty state. Flush instructions (e.g., `CLWB` and `CLFLUSHOPT`) generate a constraint that denotes that the destination is in the pending state. Non-temporal stores (e.g., x86 `MOVNT` are similar to regular stores, except their destination is immediately put into the pending state (i.e., non-temporal stores are treated as a store+flush), as non-temporal stores bypass the CPU cache but are weakly ordered (like flush instructions) and still require some form of memory fence. Global fences (e.g., `SFENCE`, `MFENCE`) add constraints to indicate that all PM cache lines are clean, whereas cache-line fences (e.g., `CLFLUSH`) add a constraint denoting that their destination is clean.

4.2 Persistency Bug Oracles

AGAMOTTO uses the persistent memory state in order to support two types of persistency bug oracles. First, AGAMOTTO provides two built-in *Universal Peristency Bug Oracles*, which check for bugs based on the patterns we identify in §3. Second, AGAMOTTO allows developers to specify custom, application-specific persistency bug oracles, which we have used to provide two oracles for the PMDK Transaction interface [20].

```
// Unflushed Bug Oracle
def check_unflushed(state):
    for pm_obj in state:
        forall cachelines in pm_obj:
            if not cacheline.is_clean:
                raise error(correctness)

// Extra flush/fence Bug Oracle
def check_extra_flush(state, cacheline):
    if cacheline in state is clean:
        raise error(performance)
def check_extra_fence(state):
    if state has no pending updates:
        raise error(performance)

// Call Oracles on instructions:
def executeInstruction(state, inst):
    if (state.terminated or state.unmapped):
        check_unflushed(state)
    if inst is flush:
        check_extra_flush(state,
                          inst.cacheline)
        // do flush
    if inst is fence:
        check_extra_fence(state)
        state.commit_pending()
```

Listing 5: Pseudo-code for Universal Persistency Bug Oracles and how they are used as AGAMOTTO explores the state space.

4.2.1 Universal Persistency Bug Oracles

AGAMOTTO provides two universal persistency bug oracles, one that detects an instance of the missing flush/fence bug pattern (indicating a correctness or performance bug), and one that detects an instance of the extraneous flush/fence bug pattern (indicating a performance bug). We sketch the algorithms in Listing 5. AGAMOTTO reports a missing flush/fence bug for each cache-line in a persistent memory object that is not clean (i.e., the constraints on the persistent state indicate that the cache-line may be dirty or pending) at the time when the persistent memory is no longer addressable (due to either `munmap` or program exit). AGAMOTTO identifies an extraneous flush/fence operation bug on any flush (e.g., `CLFLUSH`) to a cache-line which must already be pending or clean based on the constraints on the persistent state. AGAMOTTO also identifies an extraneous flush/fence bug on any fence (e.g., `SFENCE` or `MFENCE`) which has no pending flushes to mark clean. For both of these oracles, AGAMOTTO reports program location information (e.g., stack frame and source code location) for the most recent update to each cache line which violates the conditions checked by the oracle. In our evaluation (see §6), we show that these oracles do not incur any false positives across a variety of PM frameworks and applications.

4.2.2 Custom Bug Oracles

In addition to the generic bug oracles, AGAMOTTO facilitates the use of custom bug oracles. Custom bug oracles are defined separately from the application, which allows them to be versatile tools for detecting application-specific bugs. For example, a developer might use a custom oracle to validate the correct usage of PM frameworks (e.g., identifying duplicate log entries in the PMDK `libpmemlog`) or assert that certain

```
class PmemObjTxAddChecker
    : public CustomChecker {
  bool in_tx;
  // [address, address+size)
  typedef pair<ref<Expr>, ref<Expr>> TxRange;
  list<TxRange> added_ranges;

  void checkTxBegin(Function *f,
                    ExecutionState &state) {
    if (!in_tx && f->getName() == "
    pmemobj_tx_begin")
      in_tx = true;
  }

  void checkTxAdd(Function *f,
                  ExecutionState &state) {
    if (f->getName() !=
        "pmemobj_tx_add_common") return;
    // 1. Get the address from the stack.
    ref<Expr> address = f.getArgument(0);
    ref<Expr> size = f.getArgument(1)
    // 2. Get end bound
    auto r_end = address + size;
    auto new_range = TxRange(address, r_end);
    // 3. Check for overlaps.
    //    If overlap, there's a bug!
    if (overlaps(state, new_range))
      reportError(state, RedundantTxAdd);
    // 4. Add the new range.
    added_ranges.push_back(new_range);
  }

  void checkTxEnd(Function *f,
                  ExecutionState &state) {
    if (f->getName() == "pmemobj_tx_end")
      in_tx = false;
  }

public:
  PmemObjTxAddChecker(...) {...}
  // This is the entry point
  virtual void operator()(
    ExecutionState &state) override {
    checkTxBegin(getFunction(state), state);
    checkTxAdd(getFunction(state), state);
    checkTxEnd(getFunction(state), state);

    if (!in_tx) added_ranges.clear();
  }
};
```

Listing 6: An psuedo-code example of a custom oracle, designed to check for redundant PMDK transaction "adds" (i.e., redundant log updates).

structures are operated on in the correct way (e.g., checking that PM referenced as `struct foo` is only ever modified in a PMDK transaction). Custom bug oracles define a function that takes as input an explored program state (i.e., the current state of symbolic memory and variables in the program) and an instruction; after each instruction is executed within this state, AGAMOTTO calls all configured custom bug oracles. We provide two case studies on designing and implementing custom oracles, which we use to find 4 application-specific bugs that were reported by prior work and 1 new application-specific bug. Both of the custom oracles which we present are precise, i.e., they do not introduce false positives. We describe them at a high-level below, then discuss their implementation in §5.

Redundant Undo Log Oracle. This oracle checks to ensure that data does not get logged in PMDK's undo log mechanism multiple times. We show a pseudo-code example of an oracle in Listing 6. PMDK's transactional API implements an undo log which is used to back up data before it is modified—if a transaction is interrupted by a program error or a crash, the data can be recovered from the log. A misuse of this API, however, can lead to redundant entries being created in the undo log, which degrades performance. To track these errors, this oracle keeps track of transaction boundaries (`TX_BEGIN`, `TX_END`) and the memory ranges backed up in the undo log. If overlapping memory ranges are added during a single transaction, the oracle signals a performance bug. We use this oracle to reproduce the application-specific performance bug found by PMTest in PMDK's example B-tree data structure.

Atomic Operation Oracle. This oracle ensures that a developer-specified structure is crash-recoverable through correct use of a PMDK transaction. In particular, the oracle verifies that the structure is only updated within a PMDK transaction and is properly added to the PMDK undo log. We used this oracle to find 3 existing bugs; 2 in the PMDK Atomic Hashmap and 1 in Redis-pmem.

4.3 PM-Aware Search Algorithm

AGAMOTTO uses symbolic execution to explore the state space of the program. In order to analyze large persistent memory applications, AGAMOTTO prioritizes exploring program states that are most likely to modify persistent memory using a PM-aware search algorithm. We now first explain the static analysis that AGAMOTTO uses to compute exploration priorities. We then explain the operation of AGAMOTTO's state space exploration and why AGAMOTTO's approach is more effective at finding persistency bugs than traditional coverage-guided exploration heuristics.

4.3.1 Whole-Program Static Priority Computation

The goal of AGAMOTTO's static analysis is to determine the number of reachable PM-modifying instructions from each instruction in the program. That way, AGAMOTTO can guide symbolic execution towards program locations that are expected to access PM heavily, and uncover more bugs. This technique can be effective as the number of overall instructions expected to modify PM is much smaller than the number of instructions which modify volatile memory [59].

To achieve this, AGAMOTTO first identifies all PM-modifying instructions in the program by leveraging a sound, whole-program (i.e., interprocedural) pointer analysis [4, 14, 31, 32]. The analysis maps each pointer in the program to a set of memory locations; soundness guarantees that any two pointers which may alias will have a non-empty intersection of these sets of memory locations.

```
1  char *pbuf = mmap(<PM file>);
2  ...          // (# of PM-modifying insts)
3  do_read = ...                          // (2)
4  if (do_read)                           // (0)
5      a = pbuf[x]                        // (0)
6      foo()                              // (0)
7  else                                   // (2)
8      a = ...                            // (2)
9      pbuf[x] = a                        // (2)
10     clwb(pbuf[x])                      // (1)
11     // BUG: Missing sfence!
12 exit(0)                                // (0)
```

Listing 7: An example of AGAMOTTO's static analysis. All PM-modifying instructions are highlighted. Each instruction is annotated with a comment which denotes the result of the priority calculation.

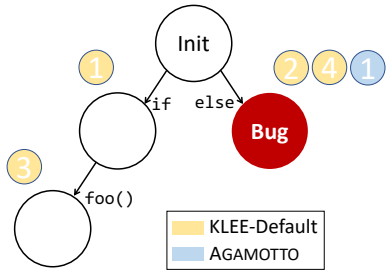


Figure 2: State space exploration with two strategies: (1) KLEE-Default (based on code coverage), (2) AGAMOTTO's priority-driven exploration. This example corresponds with the bug described in Listing 7.

AGAMOTTO then determines whether a given memory location may have been allocated as persistent memory. To do this, AGAMOTTO conservatively assumes that all `mmap` calls which accept a non-negative or variable file descriptor may return a pointer to persistent memory. While this approach over-approximates the persistent memory allocated by the program, as we show in §6, it accelerates persistency bug finding compared to default exploration strategies. Note that this conservative approach only affects the PM-aware search strategy, it does not introduce false positives in AGAMOTTO's PM state tracking.

Then, AGAMOTTO classifies each instruction in the program as a persistent memory-modifying instruction if the instruction is a global fence (e.g., `SFENCE`), or, a store (e.g., x86 `MOV`), flush (e.g., `CLWB`), or cache-line fence (e.g., `CLFLUSH`) that may point to a persistent memory location.

AGAMOTTO only computes points-to information for pointers which may alias PM. For shared libraries, AGAMOTTO first statically links the binary, then computes the alias information. If the shared library is used to modify PM (i.e., has some shared memory modification function which is used to modify PM), then that part of the shared library code will be analyzed.

Finally, AGAMOTTO uses a back-propagation algorithm to calculate the number of reachable PM modifying instructions for each program location. AGAMOTTO iterates through the interprocedural control flow graph from the exit points in the program (e.g., calls to `exit` or `return` from `main`) to the first instruction in the program. For each instruction, AGAMOTTO assigns the *priority* of the instruction to be the sum of the *weight* of the current instruction (1 if the current instruction is a PM-modifying instruction, 0 otherwise) and the maximum number of reachable PM-modifying instructions from the current instruction.

We show a small example of this priority computation in Listing 7, where each instruction is annotated with the result of the priority calculation. Each PM-modifying instruction (`pbuf[x]=a` and `clwb(pbuf[x])`) adds 1 to the priority and the priorities are backpropagated to the entry point (Line 3).

4.3.2 State Exploration Strategy

AGAMOTTO relies on an existing symbolic execution engine (KLEE [8]) to explore the possible states of the program. Symbolic execution starts with an initial program state which contains a current statement (similar to a program counter), a symbolic memory (where memory values are unknown), and symbolic inputs (e.g., an unknown `integer` value). As the program statements are symbolically executed, the symbolic execution engine simulates the effects of the program statements on symbolic inputs and memory, and updates explored program state accordingly. Moreover, the symbolic execution engine forks the explored state into two every time a branch that depends on symbolic values is encountered.

After executing a program statement in an explored state, the symbolic execution engine selects a new state to advance next. When selecting a state to explore, AGAMOTTO chooses the state whose current statement has the highest statically-computed aggregate priority (i.e., number of reachable PM modifying statements from the current instruction).

Fig. 2 shows an example of state space exploration for the the example code snippet in Listing 7, where Init represents the initial state of the program and the buggy state where the program omitted an `sfence` instruction is in the `else` path. For brevity, `foo` is depicted as a single statement that is explored at once.

The KLEE-Default strategy, which is a breadth-first exploration strategy augmented by randomized, coverage-guided prioritization, may explore states that are not useful to detecting the bug. When applied to the code in Listing 7, the KLEE-Default exploration strategy will explore the state in the `if` branch for a single statement (`a=pbuf[x]`) and switch to the state in the `else` branch for another statement (`a=...`). This cycle will repeat once more in the `if` branch (`foo()`) and in the else branch (`pbuf[x]=a, clwb(pbuf[x])`); exploration will reach the bug in a total of 4 state transitions.

AGAMOTTO, on the other hand, directly explores the `else` branch because its static analysis assigns the `else` branch a high aggregate priority. Consequently, AGAMOTTO can

discover the bug with a single state transition.

Although the number of explored states in our example is small, in practice, the number of states in a program is exponential in the number of branches that depend on symbolic input. Consequently, AGAMOTTO's exploration strategy allows it to discover many more bugs compared to KLEE's default strategy, as we demonstrate in §6.

5 Implementation

AGAMOTTO comprises a persistent memory model (~400 LOC of C++), a static analysis component (~2600 LOC of C++), and a state space exploration component (~100 LOC of C++) built atop Klee [8]). AGAMOTTO also provides 2 custom bug oracles for validating the use of the PMDK transaction API (~180 LOC of C++ for both oracles and ~200 LOC of C++ for shared custom oracle API functions).

Running real-world complex PM applications also required expanding KLEE by ~4000 LOC of C++. These additional changes were primarily to the *environment model*, which symbolically simulates syscalls and operating system facilities, such as a file system. AGAMOTTO targets the Intel x86 ISA since it is the most broadly-used platform for PM programming. Hence, AGAMOTTO adds support to KLEE for interpreting PM-specific x86 instructions (e.g., `CLWB`). Supporting a different ISA or persistency model [34, 42, 63] simply requires identifying the flush and fence operations in the ISA. In addition, AGAMOTTO adds to KLEE support for common inline assembly functions such as atomic instructions, as well as porting an extensive environment model for multithreading (i.e., POSIX threads) from Cloud9 [16], which was built on an older version of KLEE. AGAMOTTO adds support for symbolic files to model and track the state of mapped persistent memory and anonymous symbolic `mmap`. Finally, AGAMOTTO adds symbolic socket traffic to the environment model, which allows an application to receive symbolic input over a socket. Symbolic socket traffic allows AGAMOTTO to model client applications that send commands to a server process.

Developing an automated bug finding tool for persistent memory presents key challenges. To identify persistent memory allocations in a PM framework agnostic way without relying on developer annotations, AGAMOTTO tracks allocations at the system level (e.g., calls to map a persistent memory file). This represents a significant divergence from KLEE, which tracks allocations at the libc interface (e.g., `malloc` and `free`), and introduces performance challenges. Applications often allocate MBs or GBs of persistent memory, but KLEE is optimized for tracking memory objects that are KBs in size; treating each persistent memory mapping as a single memory object leads to poor performance when KLEE solves constraints. Instead, AGAMOTTO carefully partitions persistent memory into separate, yet logically adjacent, objects (empirically, we find 16KB chunks to balance the tradeoff between solver time and management overhead). AGAMOTTO also tracks the set of live persistent memory objects to reduce time resolving symbolic addresses for global fence operations.

AGAMOTTO supports custom persistency bug checkers with a simple yet powerful interface. Specifically, a developer implements a method that takes as input the state being explored symbolically and asserts pre- and post- conditions on the state of persistent memory based on an understanding of how their application should behave. AGAMOTTO provides a library of basic utilities (e.g., error reporting, calls to the symbolic solver) that comprise ~200 LOC and allows bug oracles to use type information provided by LLVM. AGAMOTTO provides 2 custom oracles to detect application-specific persistency bugs in PMDK and Redis (§4.2.2). We implement the *Redundant Undo Log Oracle* in 96 LOC and less than a day of developer effort. The *Atomic Operation Oracle* extends the Redundant Undo Log Oracle—it comprises an additional 86 LOC on top of the inherited functionality and also took less than a day to implement.

6 Evaluation

In this section, we evaluate the effectiveness and usefulness of AGAMOTTO. We start by giving an overview of the new bugs AGAMOTTO has found (84)[3] and the insights we gather from them (§6.1). We also discuss the positive responses that we have received after reporting bugs to PM application developers (§6.2). We then evaluate the performance of AGAMOTTO and how our novel search tactic compares to the default symbolic execution search strategy in KLEE (§6.3).

Evaluation Targets. We evaluate AGAMOTTO by testing representative state-of-the-art PM-application and libraries consistent with the libraries and applications tested by prior work [49, 50]. We evaluate AGAMOTTO on two PM libraries. First, we test the PMDK [20] library from Intel, the most active and well-maintained open-source PM project, which has been maintained for over 6 years. Consistent with existing tools [50], we use example data structures provided with PMDK (e.g., B-tree, RB-tree and hashmap implementations) and an application provided by Intel [22] as drivers for our testing. In addition to PMDK, we test NVM-Direct, a PM library from Oracle that is under active development. To drive our testing of NVM-Direct, we use their example test application they provide for demonstrating the API.

We additionally evaluate AGAMOTTO by testing three real-world PM applications. We test Redis-pmem, a port of Redis, a popular in-memory database and memory caching service, to PMDK that is maintained by Intel. We likewise select memcached-pm, a port of memcached, a popular high-performance memory caching server, to PMDK that is main-

[3] We provide a link to our evaluations results in the AGAMOTTO GitHub repository: https://github.com/efeslab/agamotto/blob/artifact-eval-osdi20/artifact/README.md

System	Source (GitHub)	Version
PMDK	pmem/pmdk	v1.8
RECIPE	utsaslab/RECIPE/tree/pmdk	53923cf
memcached-pm	lenovo/memcached-pmem	8f121f6
NVM Direct	oracle/nvm-direct	51f347c
Redis-pmem	pmem/pmem-redis	cc54b55
	pmem/redis	v3.2

Table 2: Software configuration; we tested two versions of Redis-pmem

tained by Lenovo. Finally we test RECIPE's P-CLHT index, a state-of-the-art persistent index representing a research prototype. Note, we only test the P-CLHT index from RECIPE because the other four indices all use a volatile allocator which prevents crash-consistency. Since KLEE symbolically emulates system calls without running real kernel code, we are unable to test PMFS [27], an evaluation target that has been considered by prior work [50].

We test each application by providing a symbolic environment model (e.g., providing symbolic arguments and files with symbolic contents) rather than instrumenting the source code to create symbolic variables. We test RECIPE's P-CLHT index using their example application, which manipulates the basic structure of the index through standard insertion, deletion, and lookup operations. We use symbolic socket traffic (See §5) to test the Redis-pmem and memcached-pm server daemons using partially symbolic packets (i.e., packets with some concrete values, like the Redis command string, with symbolic values for the keys and values).

When testing applications that use PMDK (PMDK, Redis-pmem, and RECIPE), we enable both universal bug oracles and our two custom bug oracles designed for PMDK (see §4.2.2). When testing NVM-Direct, we only use the universal bug oracles.

When using AGAMOTTO to test an application, AGAMOTTO also tracks all persistent memory use from the libraries used by the application. In the case that AGAMOTTO finds a bug in PMDK while testing an application which uses PMDK (e.g., memcached-pm, Redis-pmem, or RECIPE), we report the bug as a bug in PMDK.

Evaluation Setup. We ran our experiments across two servers, one with a Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz and one with a Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz. Each individual experiment (a single run of AGAMOTTO) was limited to a max of 10 GB of DRAM and 1 hour of runtime. We show our software configuration in Table 2. Note that none of our experiments use persistent memory hardware since AGAMOTTO symbolically models all interactions with persistent memory.

	MC		MP		EP		AS		Total	
System	N	K	N	K	N	K	N	K	N	K
memcached-pm	1	-	19	-	1	-	-	-	21	-
NVM-Direct	7	-	7	-	9	-	-	-	23	-
PMDK	1	1	14	-	6	-	1	3	22	4
RECIPE	1	-	7	-	6	-	-	-	14	-
Redis-pmem	3	-	1	-	-	-	-	1	4	1
Total	**13**	**1**	**48**	**-**	**22**	**-**	**1**	**4**	**84**	**5**

Table 3: The Bugs found using AGAMOTTO. For each bug class (**MC**: Missing flush/fence Correctness, **MP**: Missing flush/fence Performance, **EP**: Extra flush/fence Performance, and **AS**: Application-Specific), we report the number of new bugs AGAMOTTO found, **N**, and the number of bugs detected that were previously known, **K**.

6.1 Overview

We show a summary of our bug-finding results in Table 3[4]. Overall, AGAMOTTO found 84 new bugs across our 5 main test targets: 62 missing flush/fence bugs (13 correctness bugs and 48 performance bugs), 22 extra flush/fence performance bugs and 1 new application-specific correctness bug. We also detect all 5 persistency bugs found by prior work in user-space applications and confirm that we find no false positives with our universal or custom oracles. Here, we describe the bugs that we find in greater detail.

Missing flush/fence bugs. Using our built-in unflushed bug oracle, we found 62 new bugs; we manually identified that 13 are correctness bugs and 48 are performance bugs. Of the 13 correctness bugs, 10 are caused by missing flushes and 3 are caused by missing fences—all of the missing fence bugs are found in Redis-pmem. AGAMOTTO found the missing flush/fence bug in PMDK that was reported by PMTest. Of the correctness bugs, AGAMOTTO finds 1 in memcached-pm, 1 in PMDK, 1 in RECIPE's P-CLHT index, 7 in NVM-Direct, and 3 in Redis-pmem. Of the performance bugs, AGAMOTTO finds 19 in memcached-pm, 14 in PMDK, 7 in RECIPE's P-CLHT index, 7 in NVM-Direct, and 1 in Redis-pmem.

Extra flush/fence bugs. We found 22 new bugs using the extra flush/fence bug oracle. Of these bugs, AGAMOTTO found 9 in NVM-Direct, 6 in PMDK library functions and 6 in RECIPE's P-CLHT index.

Application-specific bugs. AGAMOTTO identified 1 new application-specific correctness bug in the PMDK atomic hashmap example using the extra flush/fence universal bug oracle. Using the atomic operation oracle, AGAMOTTO found all

[4]We provide the full detailed table in an online table available here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#resources.

3 application-specific correctness bugs which were reported by XFDetector[5] Using the redundant undo log oracle, AGAMOTTO detected the application-specific performance bug in the PMDK example B-tree structure that was discovered by PMTest. AGAMOTTO is unable to find the application-specific performance bug that PMTest found in PMFS because AGAMOTTO is unable to execute kernel code.

6.2 AGAMOTTO Reporting

We presented our initial results to Intel's PMDK team, Oracle's NVM-Direct team, and to the authors of RECIPE and received overall positive feedback. At the time of writing, we have not yet heard back from Lenovo developers regarding bugs in memcached-pm. PMDK developers confirmed our findings about performance issues. Oracle's developers confirmed they were aware of some of the issues we reported and noted that "Resources for software development are always in short supply, so the open source version of NVM_Direct has suffered. I wish it was not so, but it is. Your email may be the push that gets us to do something about it. Thank you." RECIPE's authors confirmed and started patching all the bugs we reported to them and asked us to open-source AGAMOTTO for continued testing. Despite existing tools for testing PM (one of which was even built for RECIPE [45]), one of RECIPE's authors stated that "These are some really good finds, since it was difficult to debug our own code without having a proper tool."

We conclude that AGAMOTTO has been successful in finding bugs that developers care about.

6.3 Performance Analysis

Benefit of AGAMOTTO's State Exploration Strategy. We evaluate AGAMOTTO's state exploration strategy compared to the default search strategy in KLEE. We compare these two strategies for all of our 5 test targets: memcached-pm (Fig. 3a), NVM-Direct (Fig. 3b), RECIPE's P-CLHT index (Fig. 3d), on PMDK's libpmemobj examples (Fig. 3c), and on Redis-pmem (Fig. 3e). We run each exploration strategy for one hour, since one hour is short enough to integrate into a development cycle but long enough to cover a substantial number of execution paths. In all cases, AGAMOTTO's search strategy finds all reported bugs in less than 40 minutes. For Redis-pmem, the bugs we detect were exposed quickly, allowing both strategies to find all 4 in under 3 minutes. For all of our tests, AGAMOTTO is able to find at least one bug in under 5 minutes, which suggests that AGAMOTTO might even be usable during interactive debugging sessions.

We conclude that AGAMOTTO's static-analysis guided search strategy is more effective in finding bugs than the default state exploration strategy in KLEE.

[5] XFDetector reports 4 new bugs, but one of these bugs is unrelated to persistent memory but detectable with their fault injection framework.

System	Source Size (KLOC)	Dependencies (KLOC)	Static Analysis Run time (min)
memcached-pm	18	36	2.20
NVM-Direct	1	14	0.02
PMDK	2	35	0.60
RECIPE	13	35	0.55
Redis-pmem	54	149	19.6

Table 4: The offline overhead of AGAMOTTO's static analysis. Thousand lines of code (KLOC) is provided for program sources (the driver applications for NVM-Direct and PMDK) and for shared libraries.

Static Analysis Run time. We show the run time of AGAMOTTO's static analysis in Table 4. For most applications we test, the overhead of static analysis is low (less than 4 minutes) relative to the length of time spent finding bugs. Redis-pmem has a larger static analysis run time, particularly due to the number of external libraries it links with—however, the results of the static analysis can be cached across many runs for external libraries.

6.4 Case Study: PM Performance Bugs

Prior works on PM argues for the importance of the performance bugs that are identified by AGAMOTTO. For example, Pelley et al. show that extra flush and fence operations are detrimental to application performance [63], and a study of memcached-pm found that storing volatile data in PM reduces application performance by roughly 5% [26].

To further validate the importance of the performance bugs identified by AGAMOTTO, we perform a performance case study on the P-CLHT data structure from RECIPE. We manually fix the performance bugs and then measure the performance of the data structure on concurrent insert operations, i.e., load operations (each thread inserts new keys into the hash table). We chose insert operations, since they stress the update path on which these bugs were found. We report the performance in Fig. 4. The overall throughput increases dramatically, ranging between 24% to 47%. The main contributor to this throughput increase is moving commonly used locks from PM to DRAM.

7 Related Work

Persistent Memory Frameworks. Crash consistency mechanisms for persistent memory have been considered for years [6, 11, 15, 18, 64]. The difficulty of designing crash-consistent programs for persistent memory has inspired many persistent memory specific crash-consistent frameworks which ease the burden on PM application developers. These frameworks either provide a library interface that can be used in standard programming languages (PMDK [20], NV-Heaps [17], LSNVMM [35]), provide lan-

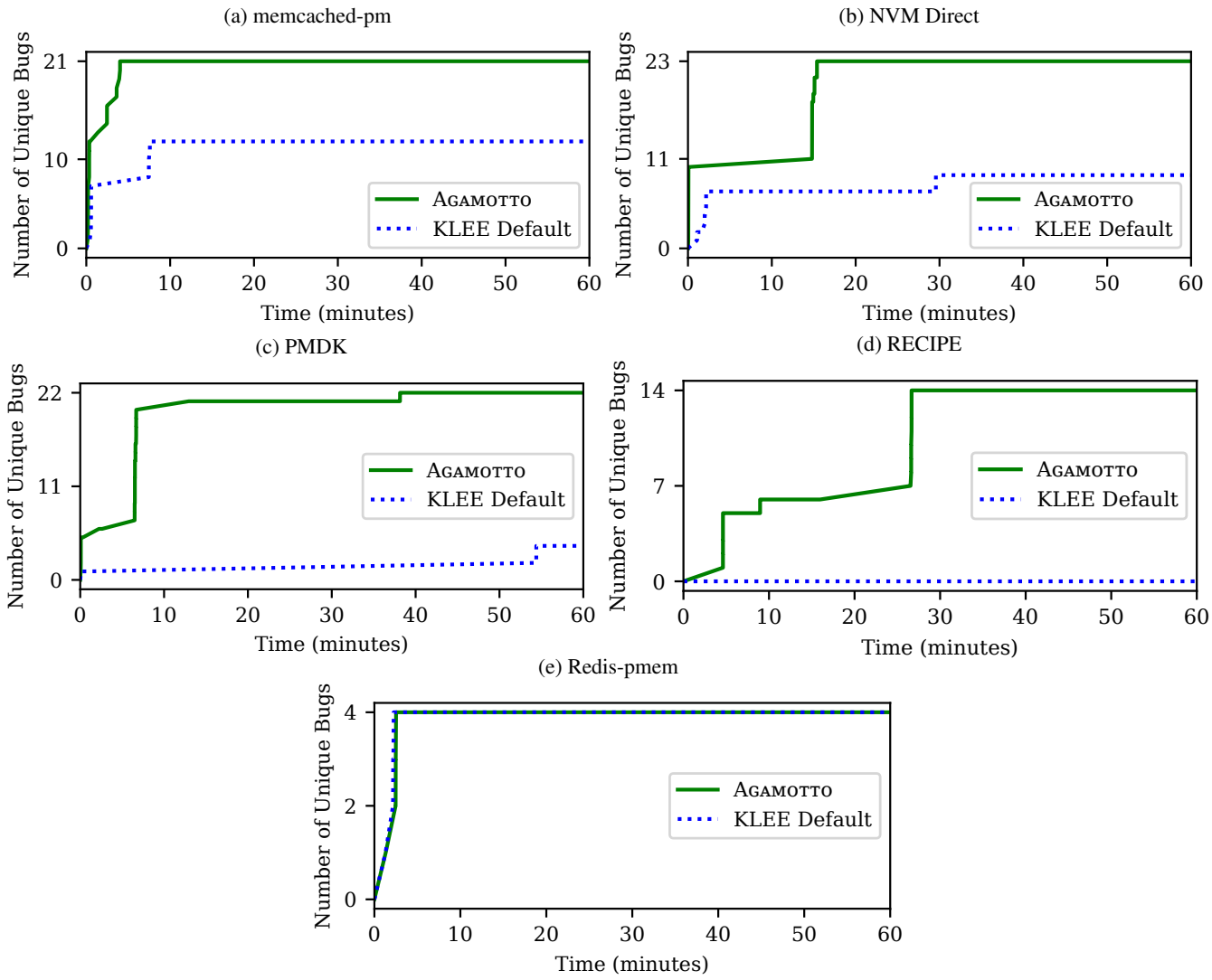


Figure 3: Comparison of the KLEE default search strategy to AGAMOTTO.

guage extensions to augment C/C++ with persistent data types (e.g., Mnemosyne [73], NVL-C [25]), or both (e.g., NVM-Direct [7]). Some systems also use transactional hardware mechanisms to provide more efficient updates to persistent memory (NV-HTM [10], Crafty [30]). However, while these mechanisms may make programming easier, they may still contain persistency bugs. Furthermore, this plethora of PM libraries and extensions motivate the need for generalizable, automated debugging tools.

PM-optimized file systems offer some degree of crash consistency as well [19,27,43,72,74,75], as many PM-optimized file systems offer full-data consistency, rather than just maintaining metadata consistency [9]. However, these mechanisms require the application to use the POSIX interface, as data journaling cannot be efficiently performed for direct-access files. Additionally, applications can suffer from significant performance degredations by acccessing PM through the file system rather than through direct memory mappings [37].

Tools for Detecting Persistency Bugs. The state-of-the-art tools for detecting persistency bugs are PMTest [50] and XFDetector [49]. PMTest is a tracing system which transforms updates to persistent memory into a trace of operations, which is asynchronously validated against programmer-defined rules for persistent memory updates. PMTest is flexible and fast, but requires developer effort to generate persistent memory rules and incurs a high rate of false negatives, as it must be driven by concrete test cases. The authors of PMTest [50] manually instrument applications to find two similar patterns to AGAMOTTO application-independent patterns: the extra flush/fence bug pattern and a delayed flush/fence pattern, in which a delay in the durability of an PM update prevents crash consistency. Delayed flush/fences are inherently application-specific (and thus require developer effort), and there were no delayed flush/fence bugs in our study. XFDetector is a fault injection framework designed to detect cross-failure bugs, which manifest when recovery code accesses

	Agamotto	PMTest	XFDetector	pmemcheck	Persistency Inspector
Core Mechanism	Symbolic Execution	Trace Validation	Fault Injection	Binary Instrumentation	Binary Instrumentation
Accuracy	High	Low	Medium	Low	Low
Automation	High	Low	Medium	Low	Low
Generality	Medium	High	Medium	Very Low	Low
Extensibility	High	High	Low	Low	Low

Table 5: A qualitative comparison between AGAMOTTO and related work, as measured by our design goals (§4).

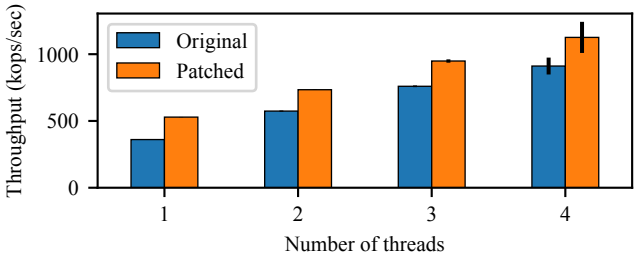


Figure 4: The write throughput (in kilo-operations per second) of the P-CLHT data structure before and after patching performance bugs. "Original" denotes the unmodified P-CLHT structure and "Patched" denotes P-CLHT after we patch the performance bugs.

data which was not guaranteed to be safely persisted before a failure. While XFDetector is effective at detecting semantic bugs with low developer effort, XFDetector still relies on developer-provided concrete test cases. RECIPE [45] uses a PIN-based tool for testing their converted PM indices, which also incurs a high false positive rate due to requiring extensive test cases. pmemcheck [65] and Persistence Inspector [62], which are binary instrumentation tools built by Intel, require a large amount of developer effort to use as they are heavily annotation based. We summarize the high-level feature differences between AGAMOTTO and other persistency bug detection frameworks in Table 5.

Tools for Testing Crash Consistency. Crash consistency testing has been the study of many works on both legacy file systems and PM-optimized file systems [13, 28, 29, 41, 44, 55, 58]. Many of these tools either test for semantic bugs specific to file systems or are only targeted for block-based storage devices. Yat [44] specifically targets crash consistency testing for Intel's persistent memory file system (PMFS [27]). However, Yat tests crash consistency by computing all possible instruction orderings to find crash consistency bugs—a task which can take over 5 years to fully test [44].

Bug Taxonomies. Many papers taxonomize software bugs in other contexts. In the storage context, JUXTA [57] draws a distinction between shallow (roughly equivalent to application-independent) and semantic (application-specific) bugs while CrashMonkey [55] studies the effects and number of operations required to induce crash consistency bugs in file systems. More generally, Li et al. [47] and Liu et al. [48] classify software bugs into universal bug classes (e.g., memory-related, concurrency and incorrect failure handling) and semantic (application-specific) bugs. The key distinction between our study and these prior studies is our focus on persistent memory systems.

The Thread Between Concurrency and Consistency. Several works have identified a similarity in data races [1, 39, 61] in concurrent programs and semantic crash consistency bugs [45, 49]. Traditional data races result in inconsistent data being read across threads of execution, which many systems have been designed to detect and fix [2, 38, 40, 46, 66, 69]. Principles from data race detection have been adapted to build PM crash consistency mechanisms (i.e., in RECIPE [45]) and PM semantic crash consistency detection tools (i.e., XFDetector [49]). When applied to AGAMOTTO, these principles inform the design of custom bug oracles.

8 Conclusion

Persistent Memory (PM) can be used by applications to directly and quickly persist data without the overhead of a file system. However, writing PM applications that are simultaneously efficient and correct is challenging. In this paper, we presented a system for more thoroughly testing PM applications. We informed our design using a detailed study of 63 bugs from popular PM projects. We then identify two application-independent (i.e., universal) patterns of PM misuse which are widespread in PM applications and can be detected automatically.

We then presented AGAMOTTO, a generic and extensible system that leverages symbolic execution for discovering misuse of persistent memory in PM applications. We introduced a new symbolic memory model that is able to represent whether or not PM state has been made persistent, as well as a state space exploration algorithm which can drive AGAMOTTO towards program locations that are susceptible to persistency bugs. We used AGAMOTTO to identify 84 new bugs in 5 different applications and frameworks, all without incurring any false positives and not requiring any source code modifications or extensive test suites.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd, Michael Swift, for their valuable feedback. We also thank Bill Bridge and the Oracle team behind NVM-Direct; Andy Rudoff and the whole PMDK team at Intel; as well as Sekwon Lee, Vijay Chidambaram, and the authors of RECIPE. This work is supported by Applications Driving Architectures (ADA) Research Center (a JUMP Center co-sponsored by SRC and DARPA), the National Science Foundation under grants CNS-1900457 and DGE-1256260, the Texas Systems Research Consortium, the Institute for Information and Communications Technology Planning and Evaluation (IITP) under a grant funded by the Korea government (MSIT) (No. 2019-0-00118), and a Microsoft Ph.D. Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

A Artifact Appendix

A.1 Abstract

We provide the public repository for AGAMOTTO, which is a fork of KLEE available on GitHub. AGAMOTTO's artifact includes instructions for building and running AGAMOTTO, as well as a pre-installed VM and scripts used to reproduce the core results from our paper.

A.2 Artifact check-list

- **Public repository link**: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact
- **Data:** Links to our bug study findings and to a table describing new bugs found with AGAMOTTO: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#resources
- **Code licenses:** AGAMOTTO inherits KLEE's open source license, which can be read in the repository here: https://github.com/efeslab/agamotto/blob/artifact-eval-osdi20/LICENSE.TXT.

A.3 Description

All information is available at our public GitHub repository. We have written a README specifically for the Artifact Evaluation process, which can be found here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact

A.3.1 How to access

We provide information on how to access our repository and all relevant resources here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#agamotto-osdi-20-artifact

A.4 Installation

The instructions for compiling AGAMOTTO and installing the prerequisites can be found here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#artifacts-functional-criteria

A.5 Evaluation and expected result

We provide instructions for reproducing the main results from our paper along with the expected results here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#results-reproduced.

A.6 Notes

We are endeavoring to maintain AGAMOTTO as an open-source tool for debugging PM applications and hope to encourage its use for a wide variety of applications. Any issues that are found with the available artifact or any needed clarifications can be submitted as GitHub issues on our repository (https://github.com/efeslab/agamotto/issues).

References

[1] Sarita V Adve and Mark D Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and distributed systems*, 4(6):613–624, 1993.

[2] Sarita V Adve, Mark D Hill, Barton P Miller, and Robert HB Netzer. Detecting data races on weak memory systems. *ACM SIGARCH Computer Architecture News*, 19(3):234–243, 1991.

[3] Paul Alcorn. Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html, 2019.

[4] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[5] Arm Limited. *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*, 2019. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

[6] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758, 2017.

[7] Bill Bridge. Nvm-direct library. https://github.com/oracle/nvm-direct, 2015.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[9] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

[10] Daniel Castro, Paolo Romano, and Joao Barreto. Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79, 2019.

[11] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[12] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMOVE: Helping programmers move to byte-based persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*, Savannah, GA, November 2016. USENIX Association.

[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, 2015.

[14] Jia Chen. Andersen's pointer analysis. https://github.com/grievejia/andersen.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243, 2013.

[16] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[17] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, 2011.

[18] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[19] Jonathan Corbet. Supporting filesystems in persistent memory, September 2014.

[20] Intel Corporation. Persistent Memory Programming. https://pmem.io/pmdk/, 2018.

[21] Intel Corporation. Redis. https://github.com/pmem/redis/tree/3.2-nvml, 2018.

[22] Intel Corporation. PMDK Examples for libpmemobj. https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj, 2020.

[23] Intel Corporation. PMDK Issues. https://github.com/pmem/pmdk/issues, 2020.

[24] Lenovo Corporation. Memcached. https://github.com/lenovo/memcached-pmem, 2018.

[25] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. Nvl-c: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 125–136, 2016.

[26] Dormondo. The Volatile Benefit of Persistent Memory: Part Two, May 2019.

[27] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[28] Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the integrity of transactional mechanisms. *ACM Transactions on Storage (TOS)*, 10(4):1–23, 2014.

[29] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage (TOS)*, 8(4):1–29, 2012.

[30] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions, 2020.

[31] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.

[32] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium*, pages 265–280. Springer, 2007.

[33] Swapnil Haria, Mark D Hill, and Michael M Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 775–788, 2020.

[34] Swapnil Haria, Sanketh Nalli, Michael M Swift, Mark D Hill, Haris Volos, and Kimberly Keeton. Hands-off persistence system (hops). In *Nonvolatile Memories Workshop*, 2017.

[35] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 703–717, 2017.

[36] Intel. Intel® Optane™ DC Persistent Memory. http://www.intel.com/optanedcpersistentmemory, 2019.

[37] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[38] Guoliang Jin, Wei Zhang, and Dongdong Deng. Automated concurrency-bug fixing. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 221–236, 2012.

[39] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. *ACM SIGPLAN Notices*, 47(4):185–198, 2012.

[40] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: crowdsourced data race detection. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 406–422, 2013.

[41] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[43] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[44] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[45] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

[46] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 162–180, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, page 25–33, New York, NY, USA, 2006. Association for Computing Machinery.

[48] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 155–162, 2019.

[49] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs.

In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1187–1202, 2020.

[50] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–425, 2019.

[51] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 216–223. IEEE, 2014.

[52] Pratyush Mahapatra, Mark D. Hill, and Michael M. Swift. Don't persist all : Efficient persistent data structures, 2019.

[53] Pratyush Mahapatra, Mark D. Hill, and Michael M. Swift. Don't persist all : Efficient persistent data structures, 2019.

[54] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[55] Ashlie Martinez and Vijay Chidambaram. Crashmonkey: A framework to systematically test file-system crash consistency. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, pages 6–6. USENIX Association, 2017.

[56] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[57] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377, 2015.

[58] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, 2018.

[59] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 135–148, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[61] Robert HB Netzer and Barton P Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[62] Kevin Oleary. How to Detect Persistent Memory Programming Errors Using Intel® Inspector - Persistence Inspector, 2018. https://software.intel.com/en-us/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.

[63] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276. IEEE, 2014.

[64] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the nvram era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.

[65] PMDK. An introduction to pmemcheck. https://pmem.io/2015/07/17/pmemcheck-basic.html.

[66] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, 2003.

[67] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: formalising the persistency semantics of armv8 and transactional models. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[68] Capegmini S.A. Capgemini world quality report 2015-2016. https://www.uk.capgemini.com/thought-leadership/world-quality-report-2016-17, 2015.

[69] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[70] Steven Swanson. Early measurements of intel's 3dx-point persistent memory dimms, Apr 2019.

[71] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pages 5–5. USENIX Association, February 2011.

[72] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[73] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[74] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[75] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[76] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.



Orchard: Differentially Private Analytics at Scale

Edo Roth, Hengchu Zhang, Andreas Haeberlen, Benjamin C. Pierce

University of Pennsylvania

Abstract

This paper presents Orchard, a system that can answer queries about sensitive data that is held by millions of user devices, with strong differential privacy guarantees. Orchard combines high accuracy with good scalability, and it uses only a single untrusted party to facilitate the query. Moreover, whereas previous solutions that shared these properties were custom-built for specific queries, Orchard is general and can accept a wide range of queries. Orchard accomplishes this by rewriting queries into a distributed protocol that can be executed efficiently at scale, using cryptographic primitives.

Our prototype of Orchard can execute 14 out of 17 queries chosen from the literature; to our knowledge, no other system can handle more than one of them in this setting. And the costs are moderate: each user device typically needs only a few megabytes of traffic and a few minutes of computation time. Orchard also includes a novel defense against malicious users who attempt to distort the results of a query.

1 Introduction

When operating a large distributed system, it is often useful to collect some data from the users' devices—e.g., to train models that will help to improve the system. Since this data is often sensitive, differential privacy [28] is an attractive choice, and several deployed systems are using it today to protect the privacy of their users. For instance, Google is using differential privacy to monitor the Chrome web browser [31], and Apple is using it in iOS and macOS, e.g., to train its models for predictive typing and to identify apps with high energy or memory usage [7, 8]. Other deployments exist, e.g., at Microsoft [27] and at Snap [68].

Today, this data is typically collected using *local* differential privacy [31]: each user device *individually* adds some random noise to its own data and then uploads it to a central entity, which aggregates the uploads and delivers the final result. This can be done efficiently at scale, but the final result contains an enormous amount of noise: as Google notes [14], even in a deployment with a billion users, it is easy to miss signals from a million users. Utility can be improved by reducing the amount of noise, but this weakens the privacy guarantee considerably, to the point where it becomes almost meaningless [80].

One way to avoid this problem is to collect the data using *global* differential privacy instead. In this approach, each device provides its raw, un-noised data to the central aggregator, which then adds random noise only once. This clearly produces results that are more precise, but it also requires a lot of trust in the aggregator, who now receives the individual users' raw data and must be trusted not to look at it. Cryptographic techniques like multiparty computation [84] and fully homomorphic encryption [38] could theoretically avoid this problem, but, at least with current technology, scaling either approach to millions of participants seems implausible.

The recently proposed Honeycrisp system [76] can provide global differential privacy at scale, with a single, untrusted aggregator. Instead of fully homomorphic encryption, Honeycrisp uses additively homomorphic encryption, which is much more efficient. However, the price to pay is that Honeycrisp can answer only one specific query, namely count-mean sketches [8] with additional use of the sparse-vector operator. This query does have important applications (for instance, it is used in Apple's iOS), but it is by no means the *only* query one might wish to ask: the literature is full of other interesting queries that can be performed with global differential privacy (e.g., [15, 31, 40, 41, 55, 64, 70, 83]). Right now, we are not aware of any systems that can answer even one of these queries at scale, using only a single, untrusted aggregator.

In this paper, we show how to substantially expand the variety of queries that can be answered efficiently in this highly distributed setting. Our key insight is that many differentially private queries have a lot more in common than at first meets the eye: while most of them transform, group, or otherwise process the input data in some complicated way, the heart of the algorithm is (almost) always a sequence of sums, each computed over some values that are derived from the users' input data. This happens to be exactly the kind of computation that Honeycrisp's collect-and-test (CaT) primitive can perform efficiently, using additively homomorphic encryption. Thus, CaT turns out to be far more general than it may seem: it can perform the distributed parts of many queries, leaving only a few smaller computations that can safely be done by the aggregator, or locally on each user device.

The key challenge is that, for many queries, the connection to sums over per-user data is far from obvious. Many differentially private queries were designed for a centralized setting where the aggregator has an unencrypted data set and

can perform arbitrary computations on it. Such queries often need to be transformed substantially, and existing operators need to be broken down into their constituents, in order to expose the internal sums. Moreover, a naïve transformation can result in a very large number of sums—often far more than are strictly necessary. Thus, optimizations are needed to maintain efficiency.

We present a system called Orchard that can automatically perform these steps for a large variety of queries. Orchard accepts centralized queries written in an existing query language, transforms them into distributed queries that can be answered at scale, and then executes these queries using a generalization of the CaT mechanism from Honeycrisp. Among 17 queries we collected from the literature, Orchard was able to execute 14; the others are not a good fit for our highly distributed setting and would require a different approach.

Our experimental evaluation of Orchard shows that most queries can be answered efficiently: with 1.3 billion users (roughly the size of Apple's macOS/iOS deployment [6]), most user devices would need only a few megabytes of traffic and a few minutes of computation time, while the aggregator would need about 900 cores to get the answer within one hour. For queries that make use of the sparse-vector operator, this is competitive with Honeycrisp; for the other queries we consider, we are not aware of any other approach that is practical in this setting. In summary, our contributions are:

- the observation that many differentially private queries can be transformed into a sequence of noised sums (Section 2);
- a simple language for writing queries (Section 3);
- a transformation of queries in this language to protocols that can answer them in a distributed setting, using only a single, untrusted aggregator (Section 4);
- the design of Orchard, a platform that can efficiently execute the transformed queries (Section 5);
- a prototype implementation of Orchard (Section 6); and
- an experimental evaluation (Section 7).

We discuss related work in Section 8 and conclude the paper in Section 9.

2 Overview

Scenario: We consider a scenario—illustrated in Figure 1—with a very large number of users (millions), who each hold some sensitive data, and a central entity, the *aggregator*, that wishes to answer queries about this data. We assume that each user has a device (say, a cell phone or a laptop) that can perform some limited computations, while the aggregator has access to substantial bandwidth and computation power (say, a data center).

Threat model: We make the OB+MC assumption from [76]—that is, we assume that the aggregator is honest-but-curious

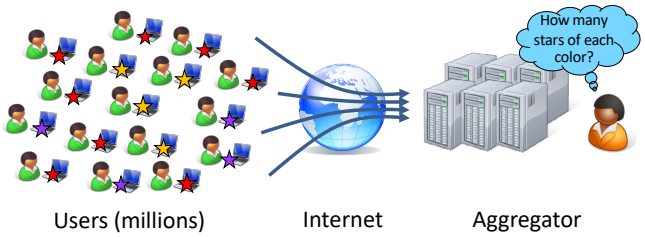


Figure 1: Scenario.

(HbC) when the system is first deployed and usually remains HbC thereafter, but may occasionally be Byzantine (OB) for limited time periods; for instance, the aggregator could be a large company that is under public scrutiny and would not violate privacy systematically, but may have a rogue employee who might tamper with the system and not be discovered immediately. For the users, we assume that most of them are correct (MC) but that a small percentage—say, 2–3%—can be Byzantine at any given time. This is different from the typical assumption in the BFT literature, where one often assumes that up to a third, or even half, of the nodes can be Byzantine. However, BFT systems are typically a lot smaller than the systems we consider: with 4–7 replicas, compromising a third of the systems means just one or two nodes, whereas, in Apple's deployment with 1.3 billion users, a 3% bound would mean 39 million malicious users, which is much larger than, e.g., a typical botnet.

Assumptions: Our key assumptions are (1) that the approximate number of users is known and (2) that the adversary cannot create and collude with a nontrivial number of Sybils. For instance, the devices could have hardware support for secure identities, such as Apple's T2 chip or Intel's SGX.

Goals: We have four key goals for Orchard:

- **Privacy:** The amount of information that either the aggregator or other users can learn about the private data of an honest user should be bounded, according to the formulation of differential privacy.
- **Correctness:** If all users are honest, the answers to queries should be drawn from a distribution that is centered on the correct answer and has a known shape;
- **Robustness:** Malicious users should not be able to significantly distort the answers; and
- **Efficiency:** Most users should not need to contribute more than a few MB of bandwidth and a few seconds of computation time per query.

2.1 Differential privacy

Differential privacy [28] is a property of randomized queries that take a database as input and return an aggregate output. Informally, a query is differentially private if changing any single row in the input database results in "almost no change" in the output. If each row represents the data of a single individual, this means that any single individual has a statistically

Query		Support
Decision-tree learning (ID3)	[34]	Yes
k-means	[15]	Yes
Perceptron	[15]	Yes
Principal Component Analysis (PCA)	[15]	Yes
Logistic regression	[2]	Yes
Naïve Bayes	[86]	Yes
Neural Network training (Grad. Descent)	[2]	Yes
Histograms	[83]	Yes
k-Medians	[40]	Yes
Cumulative Density Functions	[55]	Yes
Range queries	[45]	Yes
Bloom filters (RAPPOR)	[31]	Yes
Count Mean Sketch	[8]	Yes
Sparse vector (Honeycrisp)	[76]	Yes
Iterative Database Construction	[41]	No
Teacher Ensembles (PATE)	[64]	No
Vertex programs (DStress)	[63]	No

Table 1: Selection of differentially private queries from the literature, and support by Orchard.

negligible effect on the output. This guarantee is quantified in the form of a parameter, ε, which controls how much the output can vary based on changes to a single row. Formally, we say that q is ε-*differentially private* if, for any two databases d_1 and d_2 that differ in a single row, and any set of outputs R,

$$Pr[q(d_1) \in R] \leq e^{\varepsilon} \cdot Pr[q(d_2) \in R]$$

In other words, a change in a single row results in at most a multiplicative change of e^{ε} in the probability of any output, or set of outputs.

A standard method for achieving differential privacy for numeric queries is the *Laplace mechanism* [28], which involves two steps: first calculating the *sensitivity*, s, of the query—which is how much the un-noised output can change based on a change to a single row—and second, adding noise drawn from a Laplace distribution with scale parameter s/ε; this results in ε-differential privacy. For queries with discrete values, the standard method is the *exponential mechanism* [56], which defines a "quality score" $q(d,x)$ that measures how well a value x represents a database d, and then selects value x with probability proportional to $e^{\frac{\varepsilon q(d,x)}{2s}}$, where s is the sensitivity of q. This again results in ε-differential privacy.

Differential privacy is compositional, that is, if we evaluate two queries q_1 and q_2 that are ε_1- and ε_2-differentially private, respectively, then publishing the results from both queries is at most $(\varepsilon_1+\varepsilon_2)$-differentially private. This property is often used to keep track of the amount of private information that has already been released: we can define a *privacy budget* $\varepsilon_{\max}$ that corresponds to the maximum loss of privacy that the subjects are willing to accept, and then deduct the "cost" of each subsequent query from this budget until it is exhausted. For a detailed discussion of $\varepsilon_{\max}$, see, e.g., [46].

By now, there is a rich literature on differential privacy proposing many different forms of queries for many different use cases. We have done a careful survey to collect examples that would make sense in our highly distributed setting; Table 1 contains the queries we found, which will also be used in our evaluation (Section 7.1).

2.2 Alternative approaches

Local differential privacy (LDP): As discussed earlier, another way to avoid trusting the aggregator is to use LDP [31]—that is, for each user to add noise to his or her data individually, *before* uploading it to the aggregator, instead of noising just the final result. However, there are two important challenges. The first is that the noise in the final result now grows with the number of users: for instance, a sum of values from N users now contains N draws from a Laplace distribution $L(\frac{s}{\varepsilon})$, instead of just one! The effective error grows a bit more slowly, with $\Theta(\sqrt{N})$ [29, §12.1], but still, with $N = 10^9$ and $\varepsilon = 0.1$, the median error will be approximately 300,000 with LDP and only 10 with GDP—a difference of several orders of magnitude, which can be severely limiting in practice [14]. The second challenge is that the noise is added by the *users* and not by the aggregator; thus, even a very small number of malicious users can, by using large, correlated values as their "noise" terms, severely distort the final result [22]. We will revisit this problem in Sections 5.3 and 7.3.

Multiparty computation (MPC): In principle, the data could also be aggregated using MPC [84], a cryptographic technique that enables a group of participants to jointly evaluate a function f such that each participant only learns the final output of f, but not the inputs of each participant. It may seem that all we need to do is set $f := q \circ L(\frac{s}{\varepsilon})$, where q is the query and L is a draw from an appropriate Laplace distribution. The problem, however, is efficiency: generic MPC scales poorly with the number of participants. While there are very efficient solutions for two parties (e.g., [49]) and reasonably efficient ones for a few dozen parties (e.g., [82]), we are not aware of a technique that would be practical with millions or billions of participants.

Fully homomorphic encryption (FHE): With FHE [38], users could encrypt their data with a public key and upload them to the aggregator, who could run the query on the ciphertexts, add noise, and then decrypt only the final result using a private key. As with MPC, this approach works for arbitrary queries, and it has the advantage that most of the work is done by the aggregator. However, if the aggregator has the private key, it can also decrypt the users' individual uploads—and even if this problem were solved somehow, computation on FHE ciphertexts is still many orders of magnitude slower than computation on plaintexts, so, with a billion participants, this approach does not seem realistic.

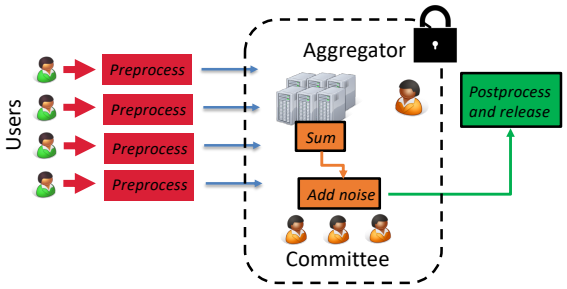


Figure 2: CaT workflow.

2.3 Honeycrisp

Honeycrisp [76] can efficiently answer one specific query (namely count-mean sketches) in our setting. As in the hypothetical FHE approach, users encrypt their private data and upload only the ciphertexts to the aggregator; however, there are two critical differences. The first is that Honeycrisp uses *additively* homomorphic encryption, which is orders of magnitude faster than FHE and can be done efficiently at scale. The second is that, to prevent the aggregator from decrypting individual ciphertexts, Honeycrisp delegates key generation and decryption to a small *committee* of 20–40 randomly selected user devices, which uses MPC to perform these (small) tasks. As before, this enables the aggregator to do all of the "heavy lifting" (collecting and aggregating ciphertexts) without ever seeing unencrypted data from individual users; thus, the aggregator does not need to be trusted.

The main drawback of Honeycrisp is that it only supports a single query. Internally, it uses a primitive called Collect-and-Test (CaT), which works roughly as follows (see also Figure 2): each user device computes a vector of numbers, encrypts it with a public key that was generated by the committee, and uploads it to the aggregator, which sums up the ciphertexts using the additive homomorphism. The aggregator then proves to the users that it has computed the sum correctly (which the aggregator, in its Byzantine phases, may not necessarily do); if so, the committee noises and decrypts the final result. This is the primitive that we leverage for Orchard.

Notice that CaT aggregates *vectors*, not just individual numbers. For additively homomorphic encryption, Honeycrisp uses Ring-LWE, which has large ciphertexts that can be subdivided into many smaller fields; these can then be aggregated in parallel. The choices from [76] yield 4,096 counters with about 50 bits each; thus, a single invocation of CaT can efficiently sum up vectors with thousands of elements. We will leverage this fact for our query optimizations (Section 4.5).

2.4 Approach and roadmap

Our key insight is that CaT is far more general than it might appear: indeed, the sums it can compute are at the heart of a wide range of differentially private queries. (This is not a coincidence: in fact, a common way to certify differential privacy—e.g., in [10,25,36,42,72,85]—is to use a linear type system to track how much a change in a single user's data can affect a given sum or count.) Thus, by rewriting queries to take advantage of CaT, we can considerably expand the range of queries that can be answered at scale. At a high level, Orchard works as follows:

1. The analyst submits her query as a *centralized* program that computes the desired answer based on a (hypothetical) giant database that contains data from all users. Orchard verifies that the query is differentially private (Section 3).
2. Orchard transforms this program into a distributed computation that relies on CaT, using several optimizations—such as vectorization—to ensure efficiency (Section 4).
3. Orchard executes the distributed program, using protocols from Honeycrisp with some additional steps, and returns the answer to the analyst (Section 5).

3 Query language

There are several existing programming languages (e.g., [10, 26, 36, 42, 57, 59, 85, 86]) that can certify differential privacy. Rather than proposing yet another, we adopt an existing language, Fuzz [42]. Fuzz is a functional language, which simplifies our transformations, and its privacy analysis is driven by lightweight type annotations, which is convenient for the analyst. However, the choice is not critical; other languages could be used as well.

3.1 Running example: k-means

To conserve space, we introduce the Fuzz language through an example: the widely used k-means clustering algorithm, shown in Figure 3, which will also be our running example for the rest of this paper. For a more complete description of Fuzz, please see [77, §A].

The k-means algorithm divides a given set of points (the input data) into k clusters and returns a centroid for each cluster. It proceeds in several iterations; for clarity, the figure shows only the iteration step, with k hard-coded to 3. The `step` function is given the current estimates of the centroid positions, `c1`, `c2`, and `c3`, and the set of points `pts`; it first assigns each point to the closest centroid, based on the l_2 distance (`assign`), and then partitions the set of points into three subsets, one for each centroid. Finally, it produces three new centroid positions `c1'`–`c3'` for the next iteration by averaging the coordinates of the points in each subset. This is done by first summing up the coordinates in each partition, and by counting the points; then the `lap` primitive adds Laplace noise to the sums and counts, and then performs the division.

3.2 Language features

In most ways, Fuzz is a conventional functional language; just two special features are relevant here. One is that it has

a *linear type system*, described in [72], that certifies an upper bound on the sensitivity of all operations on private data; when a noising primitive such as `lap` (for the Laplace distribution) or `em` (for the exponential mechanism) is invoked, the parameter s (Section 2.1) is known, and the noise can be drawn from the correct distribution. The other feature is a *probability monad* that ensures that no private data can "escape" from the program without having passed through `lap` or `em` first. Together, these features ensure that, as long as the top-level program has a type of a certain form, it is guaranteed to be differentially private.

Fuzz encapsulates private data in variables of a special type, `bag`, which represents a set with one element for each individual who contributed data. There are several primitives that operate on bags: `bmap` applies a given function to each element of a bag, `bfilter` removes elements for which a given predicate returns false, and `bpartition` splits a bag into several sub-bags, based on the value a given function returns for each element. All of these primitives take bags as arguments and produce new bags, so the private data remains confined in bags. The final bag primitive is `bsum`, which adds up the elements of a bag.

3.3 Alternative languages

Using a language other than Fuzz should not be difficult because the key to Orchard, the basic structure of summing followed by a release mechanism, is present in many other languages for differential privacy. Notice that, in Fuzz, summing via `bsum` is the only way to turn bags into data values that can potentially be released. A similar structure is present, e.g., in PINQ [57], which has three aggregation primitives, of which one (`NoisySum`) is equivalent to `bsum` followed by `lap`; the other two (`NoisyAvg` and `NoisyMed`) are equivalent to `bsum` followed by `em`. Another imperative example, Fuzzi [86], supports the addition of new aggregation primitives through an extension mechanism, but the information we need could be specified as part of the extension. The critical features Orchard needs are 1) a sensitivity analysis and 2) a way to recognize the aggregation primitives in the code.

Another possible approach would be to embed Fuzz as a library into a more traditional data analytics language, such as Python3. This embedded-language approach has already seen success in Deep Learning frameworks, such as TensorFlow [1] and PyTorch [66].

4 Query transformation

Next, we describe how Orchard transforms centralized Fuzz queries so that they can be executed in a distributed setting.

4.1 Program zones

We begin by observing that, if a Fuzz program is differentially private, it necessarily has a very specific structure and can be broken into three different "zones" (which we color-code in our example in Figure 3):

```
assign c1 c2 c3 pt =
  let d1 = sqdist c1 pt
      d2 = sqdist c2 pt
      d3 = sqdist c3 pt
  in if d1<d2 and d1<d3 then 0 else
  if d2<d1 and d2<d3 then 1 else 2

noise totalXY size = do
  let (x, y) = totalXY
  in do x' ← lap 1.0 x
        y' ← lap 1.0 y
        size' ← lap 1.0 size
  return (x'/size', y'/size')

totalCoords pts =
  let ptxs = bmap fst pts
      ptys = bmap snd pts
  in (bsum 1.0 ptxs, bsum 1.0 ptys)

countPoints pts =
  bsum 1.0 (bmap (\pt → 1) pts)

step c1 c2 c3 pts =
  let [p1, p2, p3] =
        bpartition 3 (assign c1 c2 c3) pts
      p1TotalXY = totalCoords p1
      p1Size = countPoints p1
      p2TotalXY = totalCoords p2
      p2Size = countPoints p2
      p3TotalXY = totalCoords p3
      p3Size = countPoints p3
  in do
    c1' ← noise p1TotalXY p1Size
    c2' ← noise p2TotalXY p2Size
    c3' ← noise p3TotalXY p3Size
  return (c1', c2', c3')
```

Figure 3: One step of the k-means algorithm, written in Fuzz. The colors represent the "zones" of computation.

- **Red zone** computations run directly on the data of an individual user—here, the `assign` function, which finds the closest centroid for each user's data point.
- **Orange zone** computations are performed on user data that has been aggregated but not yet noised—here, the `lap` operators, which add Laplace noise to the sums.
- **Green zone** computations involve only noised data and constants—here, the final divisions in `noise` and the parts of `iter` that set up the rest of the computation.

The Fuzz type system enforces clear boundaries between these zones: data can only pass from red to orange by aggregation (via `bsum`), and aggregate data can only pass from orange to green by noising (via `lap` or `em`). Moreover, red-zone code always operates on an *individual* element of a `bag`—that is, on data from a a single user. And lastly, none of the operations producing bags offer any way to combine multiple elements of one bag when computing an element of another bag; in other words, every element of every bag that can ever exist is derived (by filtering, partitioning, or mapping) from some single element of some bag that was initially provided as input to the top-level program.

This stratification allows us to map Fuzz programs to Honeycrisp-like computations by mapping the zones to the different parties in Figure 2. Red-zone code is executed directly by user devices; computations in this zone only need the data of one user at a time, so each user device can run it without sending any secrets anywhere. The summation at the red-to-orange boundary can be done as in Honeycrisp, by users encrypting their red-zone outputs and sending them to the aggregator, who adds them up using homomorphic addition and then passes the encrypted sum to the committee. Orange-zone code can be executed by the committee, using MPC, and the members of the committee will be able to decrypt the encrypted sums only after appropriate noise is added. Data that passes from orange to green zones must first pass through a release mechanism (`lap` or `em`) and be thus noised appropriately, so green-zone code can be safely executed "in the clear" by the aggregator itself.

The Orchard compiler uses a special operator to coordinate the mapping, summing, and releasing steps among red, orange and green zones. We call this operator `bmcs` (broadcast, map, clip and sum), and introduce it in the following subsection.

4.2 The `bmcs` operator

The operator `bmcs (b,m,c,r)` takes four parameters and behaves as follows:

- first, it *broadcasts* some public state `b` from the aggregator to the user devices;
- on each user device i, it *maps* the local private data d_i to a private vector $v_i := \mathtt{m}(\mathtt{b}, d_i)$ using the provided map function `m` (which can use the public state in its computation);
- on each user device, it *clips* the elements of v_i such that $|v_{i,k}| \leq \mathtt{c}_k$; and finally
- it *sums* all these private vectors from all client devices through homomorphic addition to compute $v := \sum_i v_i$ and returns $\mathtt{r}(v)$ using the provided *release function* `r`.

The `bmcs` operator captures the workflow of a single "round" of the distributed protocol; m is the red-zone computation for that round; r is the orange-zone computation. The clipping vector c is needed to guarantee privacy (see Section 5.3).

By rewriting a given Fuzz program to use only `bmcs` rather than the individual bag operations `bmap`, `bfilter`, `bsum`, and `bpartition`, we make its "phase-structure" explicit so that we can directly evaluate it on a Honeycrisp-like distributed platform. We next describe how Orchard does this.

4.3 Extracting dependencies

When the analyst submits a Fuzz program to Orchard, Orchard begins by reducing complex bag operations (`bpartition` and `bfilter`) into combinations of the two fundamental bag operations—`bmap` and `bsum`. A `bpartition` that splits a bag into k partitions is reduced into a `bmap` that first maps each value in the bag to a partition index, followed by k `bfilter` operations that filters out each of the individual partitions. A `bfilter` operation is reduced into a `bmap` operation that maps each value v in the bag to an optional value v'—when the filter predicate evaluates to true on v, the optional value $v' := \mathtt{Some}\, v$, otherwise $v' := \mathtt{None}$.

Orchard then normalizes the program to ensure that all variable names are unique, and that each variable is either the result of a bag operation or the result of a release mechanism (`lap` or `em`). To achieve this, Orchard freshens all variable names, and performs aggressive inlining to eliminate all other variables. Conversely, if a bag operation was originally part of an expression and did not have a name, it is given one. In the resulting normal form, programs make explicit relations between the input database, the intermediate bags and released values, and the output of the program.

Next, Orchard infers dependencies between variables by building a graph with a vertex for each unique program variable. Two vertices (u, v) are connected with a directed and labeled edge f if v is the result of running the bag operation f over u. Since the normalized program only contains two simple bag operations, the label f is either the map function supplied to some `bmap`, or the clip bound supplied to some `bsum`. Since Fuzz forbids unbounded loops over private data, this graph is acyclic. Furthermore, since both `bmap` and `bsum` take one bag variable as input and produce another bag variable as output, there is at most one edge between any two vertices in this graph. This implies the graph is in fact a directed tree, and at the root of this tree is the input bag.

This tree is a complete snapshot of the red zone computations encoded in the normalized Fuzz program. Since the dependency tree tells us how to compute any bag value given the bag variable name, we only need to keep bag variable names at their use sites. So we remove all bag operations from the normalized Fuzz program, and use the dependency tree as a reference for emitting code when a bag variable is used. We call the remaining normalized program the "core".

The core contains a mixture of orange zone and green zone computations. Since Orchard eliminates all other program variables in an earlier pass, the variables in the core must either be the result of a bag computation, or the result of a release mechanism. In particular, we call the variables that are results of bag computations "exit vertices" in the tree. (These vertices are scalar numbers, and thus cannot contain any outgoing edges, because no bag operations take scalar numbers as inputs.) By analyzing the core and inspecting the path from the input database to exit vertices, we can emit code in the `bmcs` form.

4.4 Transformation to `bmcs` form

The next step traverses the core in a forward pass, while maintaining a intermediate set S of variables. The set S is the

set of variables that are results of release mechanisms at the current program position during the forward pass.

When the traversal encounters a release mechanism (`lap` or `em`), it first compares the set of variables used in this release mechanism against S. If the set of used variables is a subset of S, then this release mechanism only adds further noise to already released data, and there is no need to invoke `bmcs`.

On the other hand, if a variable v is used in the release mechanism but is is not a member of S, then v must be the result of some bag operation. In this case, we must invoke `bmcs` to compute v and release.

Let p be the path from the input database to the variable v. Orchard now computes a map function m_p and a clip value c_p as follows. It initializes $m_p := id$ and $c_p := \infty$, then it traverses p starting from the input database. When it encounters a `bmap f`, it updates $m_p := m_p \circ f$; and when it encounters a `bsum c`, it updates $c_p := c$.

In general, a release mechanism may refer to multiple variables $v_1, \ldots, v_i$ that are results of bag operations. For each v_i, Orchard walks its corresponding path p_i to compute m_{p_i} and c_{p_i}. It then fuses these map functions and clip bounds into a new map function $m\,db = (m_{p_1}\,db, \ldots, m_{p_i}\,db)$ and a new clip bound $c = c_{p_1} +\!+ \ldots +\!+ c_{p_i}$, where $+\!+$ represents vector concatenation.

Finally, if $f(v_1, \ldots, v_i)$ is the release mechanism that uses program variables $v_1, \ldots, v_i$, we build the release function $r\,sum = f(prj_1\,sum, \ldots, prj_i\,sum)$. Here, *sum* is the aggregated vector, and each prj_i projects the corresponding value for v_i out of the aggregated vector sum.

4.5 Optimizations

The transformation process that has been described so far will calculate the correct result, but in general it will produce many redundant `bmcs` operations because it walks the core in a forward pass and emits one `bmcs` call for each release mechanism that uses private data. We can do better by observing that release mechanism calls often do not depend on each other (such as the three calls to `noise` in the k-means example) and can in fact be fused into one `bmcs` call.

Orchard exposes these optimization opportunities to the code transformation process through a simple source code rewriting step. After Orchard has inlined and normalized the input Fuzz program, but before code transformation into `bmcs`, Orchard performs local dependency analysis on release mechanism calls, using a marker combinator `par` to combine release mechanisms that have no dependency relations.

For example, the three `lap` calls in the `noise` function for the kmeans example will be rewritten into:

```
((x', y'), size') ←
  par (par (lap 1.0 x) (lap 1.0 y))
      (lap 1.0 size)
```

Since Orchard inlines the `noise` function, in fact all nine `lap` calls in the `step` function for the *k*-means example will be combined through the marker `par` combinator (there are three `lap` calls in each `noise` call, and there are three `noise` calls).

The purpose of the `par` combinator is to allow code transformation to fuse release mechanisms together just by looking at the syntax of the program under analysis. In the last phase of code transformation, when Orchard encounters a `par` combinator, it first recursively emits the map and release functions for the two arguments to `par`. Let us call these map functions m_1 and m_2, and the release functions r_1 and r_2. Next, Orchard fuses them together by creating a new map function $m\,db = (m_1\,db, m_2\,db)$, and a new release function $r\,sum = (r_1\,sum, r2\,sum)$. The clip bounds are concatenated to produce a fused clip bound. The code transformation recursively fuses the release mechanisms combined with nested `par` combinators, until finally only a single `bmcs` call is emitted for all of the combined release mechanisms.

4.6 Limitations

Our implementation currently insists that all loops in the red and orange zones terminate after a finite number of rounds, and it disallows unbounded recursion in these zones. Finite loop bounds are common in the differential privacy literature because they simplify the reasoning about the privacy cost; queries with unbounded loops, such as the PrivTree algorithm [87], tend to require more sophisticated reasoning, and thus cannot be verified by most automatic checkers. If necessary, the limit in the red zone could be replaced with timeouts and default values [42]. Notice that we *do* allow unbounded loops in the green zone, so we can still use dynamic predicates to check for convergence, e.g., in k-means clustering.

Orchard's front end relies on an existing programming language and type system, and it inherits their limitations. In particular, if a query is differentially private but the Fuzz type system cannot prove it, Orchard will reject it, and if a query's real sensitivity is s_1 but Fuzz only derives a sensitivity value $s_2 > s_1$, Orchard will use s_2. These limitations could be removed by using a different source language – e.g., one with a more advanced type system, such as DFuzz [36], or one that allows the analyst to help with the privacy proofs, such as apRHL [4].

Orchard's optimization for fusing independent release mechanisms only recognizes fusion opportunities for release mechanisms that are syntactically next to each other. Due to this simplistic nature, Orchard may miss opportunities for fusion of release mechanisms that are only revealed through a more global dependency analysis. However, in our experiments, we find that this limitation does not prevent us from emitting code with the optimal number of `bmcs` calls. We plan on improving the fusion analysis in future work.

5 Query execution

Next, we describe the platform Orchard uses to execute distributed queries once they have been transformed using the method from the previous section.

5.1 Overall workflow

Orchard implements `bmcs` using the CaT primitive from Honeycrisp [76], with three important additions: Orchard supports more than one round, it adds the broadcast step (which was not needed for Honeycrisp's one hard-coded query), and it supports more general computations on the user devices and within the committee's MPC (which Orchard needs for the red and orange zones). Protocols for sortition and verifiable aggregation (discussed below) are used verbatim, so the correctness proofs from [76] still apply. The platform consists of two components: a *server*, which runs in the aggregator's data center, and a *client*, which runs on each user's device (e.g., phone or laptop). These components operate as follows.

Setup: When an analyst wants to ask a query, she formulates it in the language from Section 3 and submits it to the server. The server typechecks the query, to verify that it is differentially private; if not, it aborts. The server then transforms the query as described in Section 4, but keeps only the code for the green zone. The server then triggers a *sortition* protocol that causes a very small, random *committee* of user devices to be elected. (As in Honeycrisp, a typical committee size is about 30–40, out of perhaps 10^9 devices.) The server sends the query to the committee, whose members perform the same transformation as the server but keep only the code for the orange zone of each `bmcs` operation, as well as the associated privacy costs ε_i. The committee runs an MPC to generate a keypair for an additively homomorphic cryptosystem, and each committee member keeps a share of the private key. The server then executes the prefix (if any) of the green-zone computation that does not involve private data.

Broadcast: When the server encounters the ith `bmcs` operation, it sends the sequence number i to the committee. The committee deducts ε_i from the privacy budget ε_{max} and, if this succeeds, signs an *execution certificate* that contains the query, the public key, and the sequence number i of the `bmcs`, and returns the certificate to the server. This certificate is needed to convince the clients that the server has "paid" the privacy cost ε_i for the specific step they are about to execute; the sequence number prevents query reexecution without charging the privacy budget again.

Map and clip: The server now distributes the certificate, along with any broadcast state in the `bmcs`, to the clients. Each client (1) verifies that the committee was elected properly, that the execution certificate is signed by the committee, and that the certificate is not a duplicate; (2) transforms the query to obtain the red-zone computation for the i^{th} `bmcs` operation; (3) executes the red-zone code on its local data; (4) encrypts the result with the public key from the certificate; and (5) uploads the result to the server, along with a zero-knowledge proof that (a) the local input was in the correct range; (b) the red zone was executed correctly; and, if $i > 1$, that (c) the client has not changed its local input since the first `bmcs` in the current query.

Sum: The server aggregates all the uploads using homomorphic addition and then publishes a Honeycrisp-style summation tree, so the clients can verify that it has included each user's data exactly once; if not, they can report the aggregator. Next, the committee performs another MPC to execute the orange-zone code (which noises and decrypts the computed aggregate) and then sends the plain-text result to the server, which uses it as the result of the `bmcs` operation and continues executing the green-zone code. If the server encounters further `bmcs` operations, it repeats the broadcast, map, clip, and sum steps for each of them.

5.2 Security: Aggregator

One key difference from Honeycrisp is that Orchard's red- and orange-zone computations are not hard-coded and must be compiled from the query instead. A naïve approach could have been to have only the server perform the transformation and to have it provide the red- and orange-zone code to the committee and to the clients, respectively. However, in this case it would have been easy for the server to, say, replace the orange zone with the identity function (to disable noising) and/or to replace the red zone with "if the user is Alice, return data $\times 10^9$, else 0" (without proper clipping).

Orchard avoids this issue by (1) having the committee and the clients compile the red and orange zones directly from the original query and by (2) including the query in the execution certificate, so that all correct participants can be sure they are part of the *same* query. Since a correct client or committee member would perform the compilation as specified, it would (correctly) reject any proposed query that was not differentially private, and it would include all the necessary elements, such as clipping and noising. A dishonest server still has control over the green zone and can run any arbitrary code there. However, it can only hurt itself by doing this: the users' privacy is guaranteed by the red and orange zones, and any data that reaches the green zone is already properly declassified.

Of course, the aggregator can misbehave in several other ways, but the compilation attack is the only one that is specific to Orchard; the others were already possible in Honeycrisp, and the defenses from Honeycrisp continue to apply. For completeness, we briefly review some key defenses below; for a complete description, please see [76, §3].

Privacy budget: A malicious aggregator could try to run more queries than the privacy budget allows. To prevent this, the budget balance is maintained by the committee. In each round, the committee checks whether the remaining privacy budget is sufficient to execute the query; if so, it signs a query

authorization certificate that includes, among other things, the remaining budget and the current round number. This certificate is sent to all user devices, which check it before uploading their responses. If the committee changes, the new members rely on the budget from the previous round's certificate.

Targeting individuals: A malicious aggregator could try to learn the private data of specific users by performing the aggregation incorrectly – perhaps by leaving out data from certain users, or by multiplying the encrypted data from other users with a large constant (which is possible in an additively homomorphic cryptosystem), or even by pretending that a single user's data is the result of the entire aggregation. To prevent this, Orchard requires the aggregator to construct a *summation tree* to prove that it has computed the aggregation correctly. Each user device checks a small portion of this tree.

Reporting channel: We assume that there is an external channel that devices can use to report the aggregator, if they should discover that the aggregator has misbehaved. Like Honeycrisp, Orchard produces evidence that the devices can use to substantiate such a report; for instance, this evidence could be posted in an online forum (Twitter, Wikipedia, ...) or it could be given to the press. In a large-scale deployment, the aggregator would typically be a large entity with a reputation to lose, so this mechanism should provide an incentive for the aggregator to follow the protocol correctly.

Collusion: If the aggregator is also the manufacturer of the user devices (which would be the case, e.g., in a deployment by Apple or Google), a malicious aggregator could try to roll out a backdoored OS version or manufacture a large number of additional devices, with which it could then collude. Here, our assumption that the aggregator is Byzantine only *occasionally* (the OB in our OB+MC assumption) is critical, because it limits the potential impact of such misbehavior.

Committee tampering: For a committee of size C, Orchard requires that $\frac{2C}{5}$ committee members are honest. With 2–3% Byzantine users, as we have assumed in Section 2, the chances of randomly sampling a committee with too many Byzantine users are miniscule; with $C = 40$, the chances of ever encountering it during a period of ten years, with one round every day, would be about 0.001%. However, a malicious aggregator could try to increase this probability by preventing honest users from participating in the sortition. To defend against this, the aggregator must maintain a Merkle tree of all the users, so that the results of the election are verifiable by all devices.

5.3 Security: Malicious clients

Another key difference from Honeycrisp is that there can be more than one `bmcs` invocation and that clients can potentially learn some information about the result of previous invocations from the broadcast step. This is not a privacy issue because the type system ensures that any broadcast state has been properly noised, but a group of malicious clients could potentially use this information in a targeted attack.

As a concrete example, suppose a large online retailer uses the k-means algorithm from Figure 3 to calculate the positions for k new shipping centers, based on the locations of their current customers; suppose, further, that a small group of users wishes to ensure that one of the centers is built in their home town. Notice that each `bmcs` broadcasts the set of centroids from the previous round. In the last round, the attackers can use this information to calculate exactly (modulo noise) what their locations would need to be to move the nearest centroid to their town and then change their inputs accordingly.

To prevent adaptive attacks like this, Orchard can optionally use verifiable computation (VC) [65] on the client side. When this is enabled, clients must upload a cryptographic commitment to their local data along with their first `bmcs` response, and they must include, with each response, a zero-knowledge proof that (a) they have executed the red-zone code correctly and (b) their initial commitment opens to the input they used in the current round. With this defense, the attackers can only choose their initial inputs. As we will show in Section 7.3, this makes a successful attack much harder.

5.4 Handling churn

A third difference is that Orchard computations with multiple `bmcs` rounds can take much longer than Honeycrisp's single-round computation. This raises two concerns: (1) the workload of the committee is somewhat higher, and (2) devices are more likely to go offline during the computation.

To address the first concern, Orchard can optionally choose a fresh committee after a few `bmcs` rounds. This requires a few more devices to serve on committees, and it adds a bit more work for the overall system because each new committee has to generate a fresh keypair, but it is safe, and it limits the work that any given committee member has to perform. To address churn in the committee, Orchard uses Shamir secret sharing to ensure that the committee can reconstruct the private key even if it has lost a few of the shares because the corresponding committee members have gone offline.

This leaves the concern that some *user* devices will leave (and others join) between rounds. This does not affect correctness, since the red zone retains no state between rounds, but it does mean that the `bmcs` sums could be computed over data from slightly different sets of users. Almost by definition, differential privacy cannot release anything that is specific to particular users, so the overall impact of individual user arrivals or departures should be small [29, §2.3.2]. The effect of higher levels of churn depends on the algorithm and on the kinds of users that are joining or leaving. For instance, consider the effect that a major power outage in a large geographic region – say, the 2003 blackout in the Northeastern U.S. [33] – would have on a query that was already in progress. If the query was choosing facility locations within the United

States, the results would be severely distorted, since it would suddenly appear as if there were no users in the Northeast at all. If, however, the query was measuring the age distribution of the users, the impact would be small, since the age distribution in the Northeast would be roughly comparable to the age distribution elsewhere.

6 Implementation

For our experiments, we built a prototype of Orchard. We used Haskell to implement the Fuzz frontend and the transformations, and Python for the backend. Our prototype generates and runs the actual red-zone and orange-zone code; for the aggregation (which would be done with millions of users in a real deployment), we benchmark the individual steps and then extrapolate the cost. Overall, our prototype consists of about 10,000 lines of code, and is publicly available [62].

Encryption: For additively homomorphic encryption, we use the Ring-LWE scheme [54]. This works over a polynomial ring $R_p := \mathbb{Z}_p[x]/(x^n+1)$, where p is a prime and n is a power of 2. The secret key is a random polynomial $s(x) \in R_p$, and the public key is a pair generated by sampling a random $a \in R_p$ and setting the public key to be $(a,b) \in R_p^2$, where $b := a \cdot s + e \in R_p$, for some "error" $e \in R_p$ chosen from an appropriate error distribution. The plaintext space is $\mathbb{Z}_q^l$, where $q, l \in \mathbb{Z}$, $l \leq n$, $q \ll p$ and $|p \mod q| \ll q$. To encrypt a vector $z \in \mathbb{Z}_q^l$, the encryptor generates a random $r \in R_p$, and computes the ciphertext $(u,v) := (a \cdot r + e_1, b \cdot r + \lfloor p/q \rceil \cdot z) \in R_p^2$. Decryption is then simply $z = round(v - u \cdot s, \lfloor p/q \rceil) / \lfloor p/q \rceil$, where $round(x,y)$ rounds each coefficient of x to the nearest multiple of y. (We assume the errors e, e_1, e_2 are sufficiently small relative to p/q.)

This encryption scheme allows us to represent our key generation and decryption protocols with a small constant number of additions and one multiplication in the polynomial ring. Moreover, it allows us to pack many 'slots' of ciphertexts into one large ciphertext, with almost no additional cost. Given our security parameter choices, this scheme yields up to 4,096 counters, each with a capacity of roughly 50 bits.

MPC: We use the SCALE-MAMBA framework [50] to implement the MPC operations for key generation and for the orange zones (Section 5.1). For key generation and decryption we used code we obtained from the authors of [76]. SCALE-MAMBA supports an arbitrary number of parties and is secure in the fully-malicious model. Operations are performed in a finite field modulo a configurable prime p, which allows for the support of both integers and floating points. This is a natural fit for our Ring-LWE encryption scheme, which also requires an integer modulus, and thus no additional modular arithmetic needs to be implemented within the MPC. In Ring-LWE, the additive homomorphism of plaintexts is modulo some integer q, where $|p \mod q| \ll q$; ideally, $p = 1 \mod q$.

Secret sharing: SCALE-MAMBA also supports Shamir secret sharing [78]. We use this to shard the private key among the k committee members in such a way that any subset of $t+1$ members can reconstruct the entire key. At the same time, t dishonest nodes cannot learn anything about the key, and $t+1$ honest nodes can detect any errors introduced by dishonest nodes. This enables Orchard to tolerate the loss of a few committee members. We modified the open-source SCALE-MAMBA source code to reconstruct the secret key automatically, if needed, using the remaining shares.

Verifiable computation: We use the zk-SNARK protocol [11] to enable clients to prove, in zero knowledge, that they have done the red-zone computation correctly, with consistent inputs (Section 5.3). For benchmarking, we used the implementation from the Pequin toolchain [67].

Security parameters: We use the LWE-estimator tool [53] of Albrecht et al. [5] to obtain concrete parameters that provide sufficient security based on the best known attacks on LWE. We chose dimensionality $n = 4096$, a 128-bit prime p, and a Gaussian error distribution with $\sigma = \frac{\sqrt{2}}{2}$ (which we approximate as the centered binomial distribution with $N = 2$ trials) in each dimension, which gives over 128 bits of security. For the verifiable aggregation, we use the same choices as Honeycrisp, namely SHA-256 hashes and RSA-2048 signatures.

7 Evaluation

Our experimental evaluation is designed to answer four high-level questions: (1) How many private queries can Orchard support? (2) How well do Orchard's optimizations work? (3) How effective are Orchard's defenses against malicious clients? And (4) what are the costs of Orchard?

7.1 Coverage

To get a sense of how many (private) queries Orchard can support, we did a careful survey of the differential privacy literature to find queries that are plausible candidates for our highly distributed setting. We collected as many different kinds of queries we could find; we excluded only a) queries that were substantially similar to ones we already had (e.g., different variants of computing CDFs), and b) queries where we simply could not imagine the data being distributed across lots of individual devices.

Table 1 (in the Overview section) shows the queries we found, as well as the papers we found them in. We then implemented each query in Fuzz, taking care to write the queries as they were presented in the papers, and not in a way that would be convenient for Orchard (e.g., with computations already grouped the way `bmcs` would require them).

We found that, out of the 17 queries we found, 14 (82%) were accepted by Orchard. The three queries that did not work were PATE [64], IDC [41], and DStress [63]. These

Query	Naïve	Optimized
ID3	$2md$	$m+1$
k-means	$3m$	$m+1$
Perceptron	$2md$	$m+1$
PCA	d^2+d	1
Logistic regression	$d+1$	2
Naïve Bayes	$2d$	2
Neural Network	$2m(d+1)$	$m+1$
Histograms	b	1
k-Medians	$3m$	m
CDF	b	1
Range queries	b	1
Bloom filters	d	1
Count Mean Sketch	d	1
Sparse vector	1	1

Table 2: `bmcs` rounds needed for each query, with and without optimizations. d is the input vector length, m the number of iterations, and b the number of buckets (see Section 7.4).

queries are not a good fit for our model. DStress operates on graphs, whereas we assume a set of per-user records. IDC is a "template algorithm" with an oracle function U, and good choices for U require functions beyond simple bag operations. PATE requires training private (un-noised) "teacher" models and then training a "student" model with noisy labels provided by the teachers. In our model, only the aggregator could play the role of PATE's teachers, but we do not trust it to see sensitive data in the clear, so we cannot express this algorithm.

Overall, our data suggests that Orchard is able to execute a wide variety of differentially private queries—even though these queries were designed for the centralized model.

7.2 Optimizations

A naïve translation of a centralized query typically results in a lot more `bmcs` invocations than necessary. To estimate how much our optimizations can help with this, we compiled each query twice, once with the full transformation and once with optimizations disabled; we then counted the `bmcs` operations in the resulting programs.

Table 2 shows our results. In most cases, our optimizations substantially reduced the number of `bmcs` rounds that were needed. (The exact reduction depends on the parameters.) Since the rounds are done sequentially (the `bmcs` calls in the green-zone code are "blocking"), and since `bmcs` accounts for almost all of a typical query's runtime, this means a much lower processing time.

We manually inspected the optimized code, looking for opportunities to further reduce the number of rounds, but could not find any. In principle, Orchard's optimizations could miss opportunities for fusing release mechanisms (Section 4.6), but this did not occur for any of the queries we tried.

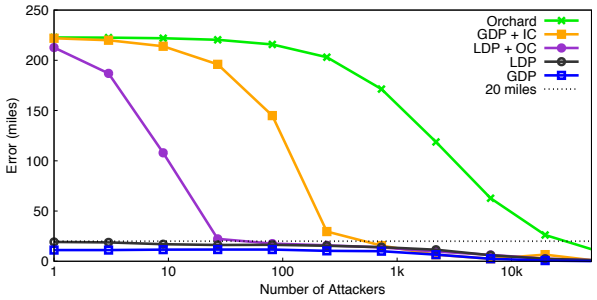


Figure 4: Impact of malicious users.

7.3 Robustness to malicious users

To examine how much Orchard's defenses help against malicious users, we implemented the attack scenario from Section 5.3. Recall that this involves an online retailer using k-means to find locations for $k = 3$ new shipping centers and a group of attackers trying to cause one of the centers to be built in their home town. We randomly sampled latitudes and longitudes for $N = 10^4$ honest users from a rectangle that includes the lower 48 U.S. states, and we used Seattle, Houston, and New York as reasonable guesses to initialize the centroid positions. We then simulated the behavior of Orchard, as well as four hypothetical alternatives: (1) local differential privacy (LDP); (2) global differential privacy (GDP) with a trusted aggregator; (3) GDP with input clipping (IC), which rejects coordinates outside the valid range and was implemented in [76]; and (4) LDP with output clipping (OC), which requires users to clip their *noised* values to $10\times$ the valid range. The attackers try to move the East Coast centroid (which is near Richmond, VA without the attack) to Pittsburgh, PA, using the strategy from Section 5.3; we assume that the attackers do not have knowledge of any data from previous Orchard queries (because, if this information was still relevant, the aggregator would likely have no need to issue a new query). We vary the number of attackers A, and we assume that the attackers are able to estimate N but do not know the locations of the other users. We say that the attack succeeds if the final East Coast centroid is within 20 miles of Pittsburgh.

Figure 4 shows the distance from Pittsburgh of the resulting East Coast centroid for each scenario and with various values for the parameters; the figure shows medians across 500 independent runs. Without a defense, GDP and LDP succumb to even a single attacker, who can observe the centroid's location in the penultimate round and then calculate an input (far outside the valid range) that will move the centroid to Pittsburgh in the final round. The residual error is due to noising; it decreases as A increases. Notice that GDP's error is even lower than LDP's; this is because GDP adds less noise.

With OC, the attackers can no longer report arbitrary values and must instead choose the largest value in the right direction that will be accepted, but the attack still succeeds with about $A = 31$ (0.3% of the users). IC further restricts the range; success now requires $A = 500$ attackers. With Orchard, the

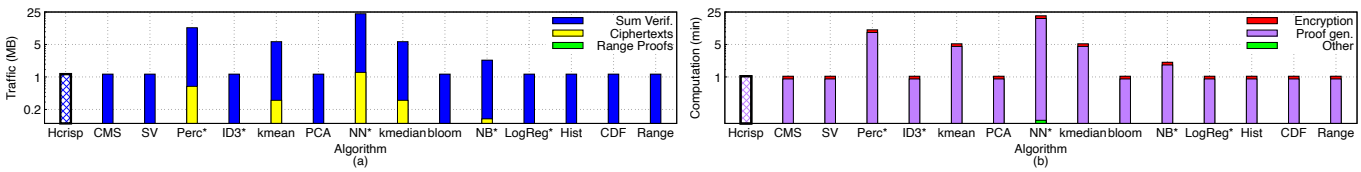


Figure 5: Bandwidth (a) and computation (b) required of each participant in a run of each algorithm.

attackers cannot adapt, and since they do not know up front what values to report—reporting, say, Portland, ME, would risk "overshooting" and moving the centroid away from Pittsburgh again—their best strategy is to simply report Pittsburgh as their location. With this strategy, the attack takes about $A = 20,000$—far more than the number of honest users.

7.4 Experimental setup

Next, we used our prototype to measure Orchard's costs to users, committee members, and the aggregator. We benchmarked the client-side software on a laptop with a 2.3 GHz dual-core processor and 8 GB of RAM running macOS Catalina. To simulate committee members operating in a global setting, we used `t2.large` EC2 instances with 8 GB of RAM, located in all available geographic regions (including the U.S., Europe, Asia, and Brazil), to get realistic latencies. For our aggregator experiments we used eight PowerEdge R430 servers with 64 GB of RAM, two Xeon E5-2620 CPUs, and 10 Gbps Ethernet; the operating system was Fedora Core 26 with a Linux 4.3.15 kernel. This equipment seems reasonably close to what a real-world aggregator might have available in its data center.

Many of our algorithms have parameters that affect the cost. For k-means and k-medians, we chose $m = 5$ and $k = 3$, because [9] notes that, given proper cluster initialization, the solution after five rounds is consistently as good or better than that found by any other method. For Perceptron, we chose $m = 10$, because the algorithm is guaranteed to converge after at most $O(1/\alpha^2)$ iterations, where α is the margin in a linearly separable dataset [75]. With vectors of size 10, we assume 1-separability to get this guarantee. For ID3, we set vector dimension $d = 100$ because we can support estimating entropy for counters of up to vectors of size 1 million (e.g., all possible 6-digit zip codes) with far fewer counters on the aggregator's side. For the neural network, we chose $m = 20$ epochs, for which [44] shows accuracy competitive with SGD.

Since Orchard is a generalization of Honeycrisp, we report Honeycrisp's numbers for comparison. We got these numbers by executing Honeycrisp's fixed query, which compiles to a single `bmcs`, with Orchard's additions disabled.

7.5 Cost for normal participants

The key costs to a normal Orchard participant are: (1) the red-zone computation itself; (2) encrypting the value to be uploaded; (3) generating the zero-knowledge proofs; and (4) verifying the aggregator's summation. (The transformations themselves are cheap; this step never took more than 410 ms for any of our 14 queries.) To quantify these costs, we benchmarked the Orchard client while it was executing each of our 14 queries; to get realistic numbers for sum verification, we emulated a system with $N = 1.3 \cdot 10^9$ users for the client to interact with. We measured the number of bytes sent, as well as the computation time spent on Orchard operations.

Figure 5 shows our results. Both the bandwidth and the computation time vary significantly between queries, but they are largely proportional to the number of `bmcs` rounds, whose cryptographic operations dominate the cost. The red-zone computations themselves are typically trivial (many simply return a value), so their cost is very small in comparison; we simply include it with the other protocol overheads in Figure 5(b). Overall, the bandwidth costs are modest, ranging from 1 MB to about 25 MB per query. The computation typically takes at most a few minutes.

The neural-network query is a an outlier; it takes about 25 minutes of computation time, which raises some concerns, e.g., about battery life on mobile devices. This high cost is mostly due to the high number of rounds we used ($m = 20$), to show what would happen when training on a "hard" problem. For "easy" lower-dimensional problems, even a single pass can be statistically optimal [69].

To measure the cost of the defense from Section 5.3, we selectively disabled the part of the zero-knowledge proof that concerns input consistency; this typically reduced the proving time by about 3%. This is because the client already has to prove that the encrypted value is in the correct range; the marginal cost of this extra proof obligation is very small.

7.6 Cost for the committee

For each query, Orchard selects a small committee of C user devices that are expected to participate in the key-generation MPC, as well as in the per-`bmcs` MPC that performs decryption and orange-zone computations. To quantify the cost to committee members, we set up committees with EC2 instances as described in Section 7.4, triggered each of our 14 queries, and measured the bandwidth and computation that the two MPCs consume. We report the cost of a single iteration of each MPC.

Figure 6 shows our results; where queries use two `bmcs` rounds per iteration, we report the cost of the more expensive one (indicated with an asterisk). The cost of the key-generation MPC depends only on the key length, and is thus identical for all queries; the cost of the orange-zone MPC

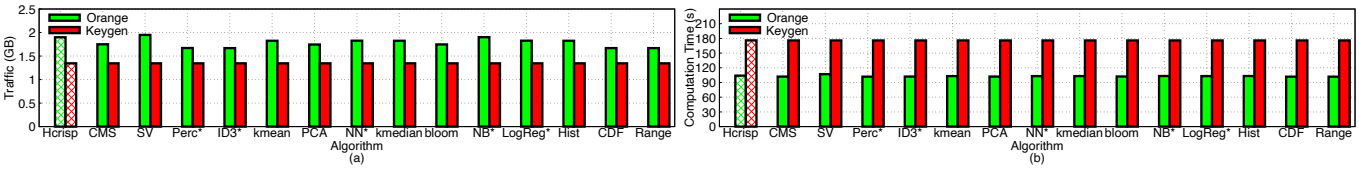


Figure 6: Bandwidth (a) and computation (b) required of each committee member during one round of orange-zone computation.

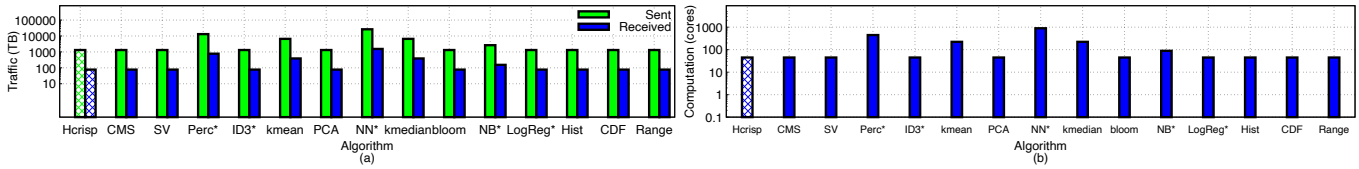


Figure 7: Bandwidth (a) and computation (b) required of the aggregator.

varies with the query, but not by much. Overall, decryption dominates the costs, and, since every `bmcs` call fits into one large packed ciphertext, we see the same behavior for all queries. In absolute terms, these costs are significant; a typical query with one round of `bmcs` consumes about 3 GB of traffic and five minutes of computation time; the total is higher if additional rounds are required.

Notice that the chances of actually being selected for the committee are tiny: for $N = 1.3 \cdot 10^9$ users, a typical committee size is about $C = 40$, so each user is only about $9\times$ more likely to be chosen than to win the jackpot in Powerball. Nevertheless, it may be useful to excuse resource-limited devices, such as mobile phones, from committee service and to rely mostly on devices like desktops and laptops, when possible.

7.7 Cost for the aggregator

Next, we quantify the costs of the aggregator, who must collect the input from each device, verify the zero-knowledge proofs, sum up the inputs, generate the summation proof, and distribute this proof to each device. We do not currently have a large enough deployment of Orchard to run this experiment end-to-end, so we estimate the costs based on benchmarks of the individual steps. We set the number of rounds as discussed in Section 7.4, and we report results for $N = 1.3 \cdot 10^9$.

Figure 7 shows the number of bytes the aggregator would need to send for each query, as well as the number of Xeon E5-2620 cores it would need to ensure that the computations do not last for more than one hour. As before, the costs depend mostly on the number of rounds; the cost of the green-zone computation is insignificant. The most expensive query (Neural Network) would require 892 cores, or 74 machines with two E5-2620 CPUs each. It would also involve sending 13,180 TB, which is a lot but actually corresponds to about 10 MB per user. For comparison: the average transfer size of a web page is about 2 MB [47]; typically, much of this is offloaded to CDNs, and the same would be possible for Orchard's summation proofs.

Scalability: We also ask how well Orchard scales with the number of participating users N. This is mostly a concern for the aggregator: the size of the MPCs (and, thus, the cost for committee members) does not depend on N at all, and the cost for individual users grows only very slowly, with $O(\log N)$, because of the summation trees. We estimate the costs of the aggregator as above, but this time we vary N.

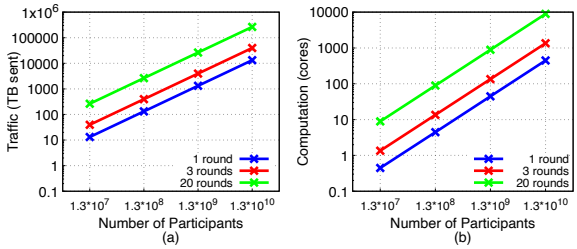


Figure 8: Bandwidth (a) and computation (b) required of the aggregator, for different system sizes.

Figure 8 shows our results (all scales are logarithmic). Although the scaling is technically $O(N \log N)$ because the height of the summation trees grows with N and each user must be sent some paths in the tree for verification, the non-linear component is small in both figures, which means that Orchard scales very well with N. This is expected, since Orchard is based on Honeycrisp, which scales similarly, and nothing in Orchard destroys this scalability.

8 Related work

To our knowledge, Orchard is the first *general* system that can process a wide variety of queries with (1) a single untrusted aggregator, (2) global differential privacy, and (3) scalability to millions of users.

Different trust assumptions: Several other systems require at least some trust in additional parties. Prochlo [14] anonymizes the user data using a shuffler, who must not collude with the aggregator; this reduces the privacy cost of LDP algorithms considerably [30]. Similarly, the crypto service provider in [37, 60] must not collude with the evaluator, and the proxy in PDDP [21] and the aggregator in Leontiadis et al. [51] must not collude with the analyst. UnLynx [35] and Prio [23] use the anytrust model, that is, a group of servers

of which at least one must be honest; SecureML [58] uses a pair of non-colluding servers; and other solutions, such as [20, 24, 48, 74], use a trusted third party for at least some steps. These additional trust assumptions yield substantial benefits, but recruiting parties that will help the aggregator but are sufficiently trustworthy to users may not be easy.

Some solutions, such as [52] use trusted hardware like Intel's SGX. We avoid this approach in Orchard because current TEE implementations are not yet sufficiently trustworthy, as shown, e.g., by the many successful attacks on SGX [61].

Local differential privacy: Google's RAPPOR [31, 32] uses LDP to aggregate data; similar systems have been deployed, e.g., by Apple [8], Microsoft [27], and Snap [68]. As discussed in Section 2.2, LDP requires significantly more noise than GDP, which can be limiting in practice [14], and it is vulnerable to attacks from small groups of colluding users [19, 22].

Smaller scale: A variety of solutions are available for systems with at most a few thousand users. For instance, Shi et al. [79] use a distributed key generation scheme to remove trust in the aggregator, and [3] use pairwise blinding instead of encryption, but these approaches do not work well under churn. Some systems have scaled MPC to impressive sizes – for instance, SEPIA [18] handles hundreds of users, and Reyzin et al. [73] perform secure aggregation for thousands, by adding homomorphic threshold encryption – but supporting millions of users with MPC seems unrealistic. Bonawitz et al. [17] use secret sharing, but, with n users, several costs grow with $O(n^2)$; Bindschaedler et al. [13] and Goryczka and Xiong [39] require $O(n^2)$ communication; Rastogi and Nath [71] use (t,n)-threshold encryption; and Halevi et al. [43] have $O(n)$ latency, since users must interact with the aggregator sequentially.

Federated learning: FL [12, 16] is another approach to working with highly distributed data. Most existing systems do not guarantee differential privacy, and the ones that do typically rely on LDP, such as [2]. Zhu at al. [88] recently proposed an interactive protocol with better privacy, specifically for discovering heavy hitters, but it does trust the aggregator with one simple task (thresholding). Truex et al. [81] relies on threshold Paillier, but it is limited to small deployments.

9 Conclusion

Prior to Orchard, it may have seemed that running differentially private queries at scale required either making compromises (on privacy, accuracy, or trust) or custom-building a cryptographic protocol. Orchard shows that, because of structural similarities among many queries, general solutions do exist, even when there is only a single, untrusted aggregator. There are still types of queries that Orchard does not support—one interesting example are queries on graphs—but we speculate that, by finding and exploiting similar structural patterns, solutions could be built for some of them as well.

Acknowledgments

We thank our shepherd Bryan Parno and the anonymous reviewers for their thoughtful comments and suggestions. This work was supported in part by NSF grants CNS-1955670, CNS-1733794, CNS-1703936, CNS-1563873, and CNS-1513694, as well as by a Google Faculty Research Award.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Whitepaper; software available from tensorflow.org.

[2] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proc. CCS*, 2016.

[3] G. Ács and C. Castelluccia. I have a dream! (DiffeRentially privatE smArt Metering). In *Proc. International Conference on Information Hiding (IH)*, 2011.

[4] A. Albarghouthi and J. Hsu. Synthesizing coupling proofs of differential privacy. *Proc. POPL*, 2017.

[5] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptography*, 9:169–203, 2015.

[6] Apple. Apple reports first quarter results. Press release, February 2018; `https://www.apple.com/newsroom/2018/02/apple-reports-first-quarter-results/`.

[7] Apple. Differential privacy. `https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf`.

[8] Apple Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 1(8), Dec. 2017.

[9] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. In *Proc. VLDB Endowment*, 2012.

[10] G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-order approximate

relational refinement types for mechanism design and differential privacy. In *Proc. POPL*, 2015.

[11] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proc. USENIX Security*, 2014.

[12] A. Bhowmick, J. Duchi, J. Freudiger, G. Kapoor, and R. Rogers. Protection against reconstruction and its applications in private federated learning. *arXiv:1812.00984 [cs, stat]*, Dec. 2018.

[13] V. Bindschaedler, S. Rane, A. E. Brito, V. Rao, and E. Uzun. Achieving differential privacy in secure multiparty data aggregation protocols on star networks. In *Proc. CODASPY*, Mar. 2017.

[14] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proc. SOSP*, 2017.

[15] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *Proc. PODS*, 2005.

[16] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. M. Kiddon, J. Konecny, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards federated learning at scale: System design. In *Proc. SysML*, 2019.

[17] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical Secure Aggregation for Federated Learning on User-Held Data. *arXiv:1611.04482 [cs, stat]*, Nov. 2016.

[18] M. Burkhart, M. Strasser, D. Many, and X. A. Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proc. USENIX Security*, 2010.

[19] X. Cao, J. Jia, and N. Z. Gong. Data poisoning attacks to local differential privacy protocols, 2019. arXiv: 1911.02046 [cs.CR].

[20] T.-H. H. Chan, E. Shi, and D. X. Song. Privacy-preserving stream aggregation with fault tolerance. In *Proc. FC*, 2012.

[21] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proc. NSDI*, 2012.

[22] A. Cheu, A. Smith, and J. Ullman. Manipulation attacks in local differential privacy, 2019. arXiv: 1909.09630 [cs.DS].

[23] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proc. NSDI*, Mar. 2017.

[24] G. Danezis, C. Fournet, M. Kohlweiss, and S. Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proc. SEGS*, 2013.

[25] A. A. de Amorim, M. Gaboardi, E. J. Gallego Arias, and J. Hsu. Really natural linear indexed type checking. In *Proc. IFL*, 2014.

[26] A. A. de Amorim, M. Gaboardi, J. Hsu, and S. Katsumata. Probabilistic relational reasoning via metrics. In *Proc. LICS*, 2019.

[27] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *Proc. NIPS*, 2017.

[28] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, 2006.

[29] C. Dwork and A. Roth. *The Algorithmic Foundations of Differential Privacy*. NOW Publishers, 2014.

[30] U. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, K. Talwar, and A. Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *Proc. SODA*, 2019.

[31] U. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proc. CCS*, 2014.

[32] G. Fanti, V. Pihur, and U. Erlingsson. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *arXiv:1503.01214 [cs]*, Mar. 2015.

[33] P. Fox-Penner. A year later, lessons from the blackout. *The New York Times*, Aug. 2004. `https://www.nytimes.com/2004/08/15/opinion/a-year-later-lessons-from-the-blackout.html`.

[34] A. Friedman and A. Schuster. Data mining with differential privacy. In *Proc. SIGKDD*, 2010.

[35] D. Froelicher, P. Egger, J. S. Sousa, J. L. Raisaro, Zhicong Huang, C. Mouchet, B. Ford, and J.-P. Hubaux. UnLynx: A Decentralized System for Privacy-Conscious Data Sharing. In *Proc. PETS*, Oct. 2017.

[36] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *Proc. POPL*, Jan. 2013.

[37] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. Privacy-preserving distributed linear regression on high-dimensional data. In *Proc. PETS*, 2017.

[38] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. STOC*, 2009.

[39] S. Goryczka and L. Xiong. A comprehensive comparison of multiparty secure additions with differential privacy. *IEEE Transactions on Dependable and Secure Computing*, 14(5):463–477, 2017.

[40] A. Gupta, K. Ligett, F. McSherry, A. Roth, and K. Talwar. Differentially private combinatorial optimization. In *Proc. SODA*, 2010.

[41] A. Gupta, A. Roth, and J. Ullman. Iterative constructions and private data release. In *Proc. TCC*, 2012.

[42] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. USENIX Security*, Aug. 2011.

[43] S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *Proc. CRYPTO*, 2011.

[44] M. Hardt, B. Recht, and Y. Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *Proc. ICML*, 2016.

[45] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially private histograms through consistency. *PVLDB*, 3:1021–1032, 2010.

[46] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *Proc. CSF*, July 2014.

[47] HTTP Archive. Report: Page weight. `https://httparchive.org/reports/page-weight`, 2020.

[48] M. Joye and B. Libert. A Scalable Scheme for Privacy-Preserving Aggregation of Time-Series Data. In *Proc. FC*, 2013.

[49] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proc. USENIX Security*, 2012.

[50] KU Leuven COSIC. SCALE-MAMBA. `https://github.com/KULeuven-COSIC/SCALE-MAMBA`.

[51] I. Leontiadis, K. Elkhiyaoui, M. Önen, and R. Molva. PUDA - Privacy and unforgeability for data aggregation. In *Proc. CANS*, 2015.

[52] D. Lie and P. Maniatis. Glimmers: Resolving the privacy/trust quagmire. *Proc. HotOS*, 2017.

[53] LWE estimator tool. `https://bitbucket.org/malb/lwe-estimator/`, commit 3019847.

[54] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Proc. EUROCRYPT*, 2010.

[55] F. McSherry and R. Mahajan. Differentially-private network trace analysis. In *Proc. SIGCOMM*, 2010.

[56] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proc. FOCS*, 2007.

[57] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proc. SIGMOD*, 2009.

[58] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017.

[59] J. P. Near, D. Darais, C. Abuah, T. Stevens, P. Gaddamadugu, L. Wang, N. Somani, M. Zhang, N. Sharma, A. Shan, and D. Song. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. In *Proc. OOPSLA*, 2019.

[60] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. *Proc. IEEE Symposium on Security and Privacy*, 2013.

[61] A. Nilsson, P. Nikbakht Bideh, and J. Brorsson. A survey of published attacks on Intel SGX. Available from `https://portal.research.lu.se/portal/files/78016451/sgx_attacks.pdf`, 2020.

[62] Orchard codebase. `https://github.com/edoroth/orchard`.

[63] A. Papadimitriou, A. Narayan, and A. Haeberlen. DStress: Efficient differentially private computations on distributed data. In *Proc. EuroSys*, Apr. 2017.

[64] N. Papernot, M. Abadi, U. Erlingsson, I. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *Proc. ICLR*, 2017.

[65] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[66] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NIPS*. 2019.

[67] Pequin: An end-to-end toolchain for verifiable computation, SNARKs, and probabilistic proofs. `https://github.com/pepper-project/pequi`.

[68] V. Pihur, A. Korolova, F. Liu, S. Sankuratripati, M. Yung, D. Huang, and R. Zeng. Differentially-private "draw and discard" machine learning. *ArXiv*, abs/1807.04369, 2018.

[69] L. Pillaud-Vivien, A. Rudi, and F. Bach. Statistical optimality of stochastic gradient descent on hard learning problems through multiple passes. In *Proc. NeurIPS*, 2018.

[70] W. Qardaji, W. Yang, and N. Li. Understanding hierarchical methods for differentially private histograms. *Proc. VLDB Endow.*, 6(14):1954–1965, Sept. 2013.

[71] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proc. SIGMOD*, 2010.

[72] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proc. ICFP*, 2010.

[73] L. Reyzin, A. Smith, and S. Yakoubov. Turning HATE Into LOVE: Homomorphic Ad Hoc Threshold Encryption for Scalable MPC. `https://eprint.iacr.org/2018/997`, 2018.

[74] E. Rieffel, J. Biehl, W. van Melle, and A. J. Lee. Secured histories: computing group statistics on encrypted data while preserving individual privacy, 2010. arXiv:1012.2152 [cs.CR].

[75] S. Rogers and M. A. Girolami. A first course in machine learning. In *Chapman and Hall / CRC machine learning and pattern recognition series*, 2011.

[76] E. Roth, D. Noble, B. Hemenway Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proc. SOSP*, Oct. 2019.

[77] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. Orchard: Differentially private analytics at scale. Technical Report MS-CIS-20-06, Department of Computer and Information Science, University of Pennsylvania, 2020.

[78] A. Shamir. How to share a secret. *CACM*, 22(11):612–613, 1979.

[79] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. X. Song. Privacy-preserving aggregation of time-series data. In *Proc. NDSS*, 2011.

[80] J. Tang, A. Korolova, X. Bai, X. Wang, and X. Wang. Privacy loss in Apple's implementation of differential privacy on MacOS 10.12, 2017. arXiv:1709.02753 [cs.CR].

[81] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou. A hybrid approach to privacy-preserving federated learning. In *Proc. 12th ACM Workshop on Artificial Intelligence and Security*.

[82] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In *Proc. CCS*, 2017.

[83] J. Xu, Z. Zhang, X. Xiao, Y. Yang, and G. Yu. Differentially private histogram publication. In *Proc. ICDE*, 2012.

[84] A. Yao. Protocols for secure computations. In *Proc. FOCS*, 1982.

[85] D. Zhang and D. Kifer. LightDP: Towards automating differential privacy proofs. In *Proc. POPL*, 2017.

[86] H. Zhang, E. Roth, A. Haeberlen, B. C. Pierce, and A. Roth. Fuzzi: A three-level logic for differential privacy. In *Proc. ICFP*, Aug. 2019.

[87] J. Zhang, X. Xiao, and X. Xie. Privtree: A differentially private algorithm for hierarchical decompositions. In *Proc. SIGMOD*, 2016.

[88] W. Zhu, P. Kairouz, B. McMahan, H. Sun, and W. Li. Federated heavy hitters discovery with differential privacy, 2019. arXiv:1902.08534.



Achieving 100Gbps Intrusion Prevention on a Single Server

Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, Justine Sherry
Carnegie Mellon University

Abstract

Intrusion Detection and Prevention Systems (IDS/IPS) are among the most demanding stateful network functions. Today's network operators are faced with securing 100Gbps networks with 100K+ concurrent connections by deploying IDS/IPSes to search for 10K+ rules concurrently. In this paper we set an ambitious goal: Can we do all of the above in a single server? Through the Pigasus IDS/IPS, we show that this goal is achievable, perhaps for the first time, by building on recent advances in FPGA-capable SmartNICs. Pigasus' design takes an FPGA-first approach, where the majority of processing, and all state and control flow are managed on the FPGA. However, doing so requires careful design of algorithms and data structures to ensure fast common-case performance while densely utilizing system memory resources. Our experiments with a variety of traces show that Pigasus can support 100Gbps using an average of 5 cores and 1 FPGA, using 38× less power than a CPU-only approach.

1 Introduction

Intrusion Detection and Prevention Systems (IDS/IPS) are a critical part of any operational security deployment [39, 40]. Such systems scan packet headers and payloads to check if they match a given set of *signatures* containing a series of strings and regular expressions. Signature rulesets are obtained through offline techniques (*e.g.*, crafted by experts or obtained from proprietary vendor algorithms) [6].

A recurring theme in IDS/IPS literature is the gap between the workloads they need to handle and the capabilities of existing hardware/software implementations. Today, we are faced with the need to build IDS/IPSes that can support line rates on the order of *100Gbps* [14] with *hundreds of thousands* [11] of concurrent flows and capable of matching packets against *tens of thousands of rules* [6]. This paper answers this challenge with the Pigasus FPGA-based IDS/IPS which meets the above goal *within the footprint of a single server*.

An important technology push that enables our effort is the emergence of server SmartNICs [29, 31]. Here, FPGA capabilities have become embedded in commodity server NICs. Of the various classes of high-performance accelerators (*e.g.*, GPUs), SmartNIC FPGAs are an especially promising alternative in terms of cost-performance-power tradeoffs, if they can be harnessed appropriately. Indeed, recent efforts have demonstrated the promise of FPGAs for low power, low costs and high performance for some simpler network functions such as software switching in Microsoft's AccelNet [21].

While many before us have integrated FPGAs with IDS/IPS processing [10, 16, 18, 19, 22, 34, 41–43, 46, 48, 49], for the most part these have focused on offloading only a specific functionality (*e.g.*, regular expression matching) to the FPGA. Unfortunately, traditional offloading cannot close the order-of-magnitude performance gap to offer 100Gbps IDS/IPS processing within the footprint of a single server. Even if regular expression search were *infinitely fast*, Snort 3.0 would still on average operate at 400 Mbps/core, requiring 250 cores to keep up with line rates! For orders of magnitude improvements, an accelerator has to improve performance for a *majority* of processing tasks, not just a small subset.

Hence in designing *Pigasus*, we argue for an *FPGA-first architecture in IDS/IPS processing*. Here, the majority of packet processing for IDS/IPS is performed via a highly-parallel datapath on-board the 100Gbps SmartNIC FPGA. Pigasus FPGA performs TCP reassembly directly on the FPGA so that it can immediately apply exact string matching algorithms over payload data to determine which "suspicious" packets need to enter a "full match" mode requiring additional string matches and regular expression matching. Inverted from the classic FPGA offload paradigm, Pigasus FPGA leaves to the CPU only the final match stage to check a small number of signatures on excerpts of the bytestream (on average, 1.1 signatures/packet and 5% of packets are sent to the CPU). By processing most benign traffic on the FPGA, the Pigasus FPGA-first architecture can reach 100Gbps and 3μs latency in the common case.

A natural consequence of Pigasus' FPGA-first approach is that we are now faced with supporting stateful packet processing on FPGA. The challenge includes not only multi-string pattern matching for *payload matching* but also *TCP bytestream reassembly* to be both performed at 100Gbps line rate. (Out-of-order TCP packets must be reassembled so detection is made even when a rule's pattern match across TCP packet boundaries.) Existing NF-specific programming frameworks for FPGAs [30, 36] do not provide the necessary abstractions for searching bytestreams, nor do they scale to the necessary scale and efficiency to meet our goals. A practical system needs to be able to track at least 100K flows and check at least 10K patterns at 100Gbps line rates, all the while staying within the available processing and memory resources of modern SmartNIC FPGAs. To meet these objectives, our design makes two key contributions:

Hierarchical Pattern Matching: Traditional streaming string search algorithms use NFA-based (state machine) algorithms. While these algorithms are very fast with small rulesets, we measured that supporting the Snort Registered Ruleset [6]

would require 23MB of Block RAM (BRAM), more than the entire capacity of our FPGA (16 MB). We instead take inspiration from Hyperscan [47], designed for x86 processors, which uses hash table lookups instead of a traditional state machine approach for exact-match string search. The (software) pattern matcher in Snort 3.0, which uses Hyperscan, offers a better starting point for our hardware design: it can support all 10K rules using 785KB of memory at a rate of 3.2Gbps on our FPGA. Scaling this to 100Gbps, however, requires replicating the state 32 times over (once again overflowing memory); Pigasus uses a set of *hierarchical filters* to reduce the overall amount of memory required per pipeline replica, enabling search over 32 indices in parallel while consuming only about 2MB of BRAM. Because Pigasus' pattern matcher is so memory-efficient, Pigasus additionally checks for extra strings in the pattern matcher that Snort would push to the 'full match' stage (implemented in Pigasus on the CPU). At the additional cost of 1.3MB of memory, scanning for extra strings results in 2× fewer packets (and 4× fewer rules/packet) reaching the full matcher than Snort.

Fast Common-Case Reassembly: Conventional approaches to TCP reassembly use fixed-length, statically allocated buffers. This prevents highly out-of-order flows from hindering performance (since insertion is constant time) and from consuming too much memory (since buffer sizes are fixed). However, fixed buffers are very memory inefficient; the strategy proposed in [49] would require 6.4GB of memory to support 100K flows. A linked list would be more memory dense, but more vulnerable to out-of-order flows stalling pipeline parallelism or exhausting buffer space. Pigasus adopts the memory-dense linked list approach, but only on an out-of-order *slow path* which runs in parallel to the primary fast path; if the memory buffer approaches capacity, large flows are preferentially reset to prevent overload. On the fast path, packets access a simple table storing the next byte expected and increment it accordingly, thus, in-order flows are performance-wise isolated from out-of-order flows. This allows Pigasus to be memory-efficient (requiring on average 5KB per out-of-order flow) while isolating well-behaved traffic from performance degradation when the IDS is inundated with misbehaving, out-of-order packet data.

Together, the above design choices allow us to do *the majority of processing on a highly-parallelized FPGA fast-path* while fitting *memory within the available resources*. As a result, for the empirical traces, Pigasus can process 100Gbps using an average of 5 cores and 1 FPGA, requiring an average of 85 Watts. In contrast, Snort 3.0 [5] – which uses Hyperscan [47] for string matching – would require 364 cores and 3,278 Watts. Pigasus is publicly available at https://github.com/cmu-snap/pigasus.

2 Background & Motivation

We now introduce software IDS/IPS systems (§2.1) and FPGAs (§2.2). We then analyze IDS/IPS performance and bound the throughput gains achievable via offloading using measurements of Snort 3.0 (§2.3).

2.1 IDS/IPS Functionality

The key goal of a signature-based[1] IDS/IPS system is to identify when a network flow triggers any of up to tens of thousands of signatures, also known as *rules*.

A given signature may specify one or several *patterns* and the entire signature is typically triggered when all patterns are found. Patterns come in the following three categories:

- *Header match*: a filter over the flow 5-tuple (*e.g.*, 'all traffic on port 80', 'traffic from 145.24.78.0/24');
- *String match*: an exact match string to detect within the TCP bytestream or within a single UDP packet;
- *Regular expression*: a regular expression to detect within the TCP bytestream or within a single UDP packet.

Signatures are detected at the granularity of a 'Protocol Data Unit' (PDU) – that is, a signature is only triggered if all matches are found within the same PDU (not over the course of the entire flow). By default, a PDU consists of one packet, but it is possible to define other protocol-specific PDUs spanning multiple packets (*e.g.*, one HTTP GET request).

When an IDS/IPS operates in *detection* mode, a triggered signature results in an alert or an event recorded to a log. When an IDS/IPS operates in *prevention* mode, a triggered signature may raise alerts, record events, or *block* traffic from the offending flow or source. IPSes hence must operate inline over traffic and are latency sensitive – a packet may not be released to the network until after the IPS has completed scanning it. IDSes, on the other hand, may operate asynchronously and are often deployed over a secondary traffic 'tap' which provides copies of the active traffic.

Software IDS/IPS Performance: One of the most widely-known IDS/IPSes is Snort [38] and our work aims to be compatible with Snort rulesets. In our experiments, we primarily work with the Snort Registered Ruleset, which contains roughly 10,000 signatures [6]. This ruleset, combined with conversations with system administrators, sets our goal of supporting 10K rules. In addition, we target 100Gbps as the state-of-the-art line rate [14] and we aim to support 100K flows. To the best of our knowledge there exists no measurement study detailing how many flows to expect at 100Gbps so we derive our 100K flow goal by extrapolating a two-orders-of-magnitude growth factor from a 2010 study [11].

In 2019, Intel published Hyperscan [47], an x86-optimized library for performing both exact-match and regular-expression string matching. Hyperscan is the key new element

[1] There exist other models of IDS/IPS which are 'script based' – executing arbitrary user code over scanned traffic – such as Zeek [7, 35]. These IDS/IPSes are out of scope for this work.

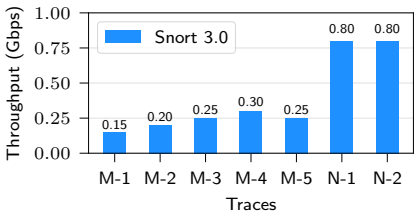


Figure 1: Single-core, zero loss throughput for Snort 3.0 over empirical traces.

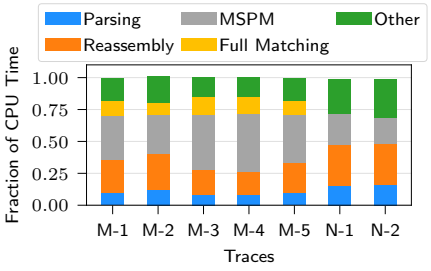


Figure 2: Fraction of CPU time spent performing each task in Snort 3.0.

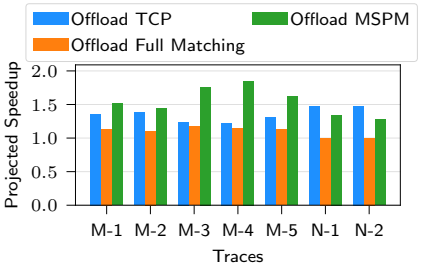


Figure 3: Projected speedup in software given various single-task offloads.

in *Snort 3.0*, which is 8× faster than its predecessor. Nonetheless, we find that Snort 3.0 cannot meet our goal of supporting 100Gbps, 100K flows, and 10K rules on a single server.

We ran Snort 3.0 on a 3.6GHz server and measured the *single-core throughput* over 7 publicly available network traces (described more thoroughly in §6.1). We plot the results in Figure 1. Generously assuming that Snort 3.0 is capable of *perfect* multicore scalability, this would require 125-667 cores to support 100Gbps of throughput, or 4-21 servers.

2.2 FPGA Basics

Why look to FPGAs to improve IDS/IPSes? While there are many platforms ('accelerators') that offer highly parallel processing, we choose FPGAs because they are (a) energy-efficient (using 4-5× fewer Watts than GPUs [12]) and (b) because they are conveniently deployed on SmartNICs where they are poised to operate on traffic without PCIe latency or bandwidth overheads.

FPGA Compute: FPGAs allow programmers to specify custom circuits using code. However, implemented naively, FPGA-based designs can be much slower than their CPU counterparts because FPGA clock rates operate 5-20× slower than traditional processor clock rates. To achieve performance speedups relative to CPUs, circuits must be designed with a high degree of parallelism. FPGAs achieve parallelism either through *pipeline parallelism*, in which *different* modules operate simultaneously over different data, or through *data parallelism* in which copies of the *same* module are cloned to operate simultaneously over different data.

FPGA Memory: Today's FPGAs offer programmers a collection of memory options. Block RAM (BRAM) is the 'king' of FPGA memory because read requests receive a response within one cycle. Furthermore, BRAM is very friendly to parallelism. Divided into 20Kb *blocks* with two ports each, it is possible to read from all BRAM blocks in parallel (and each BRAM block twice) per cycle. When a developer wishes to issue more than two parallel reads to a BRAM block per cycle, they may choose to *replicate* the block to allow more simultaneous read-only accesses to stored data. Our FPGA offers 16MB of BRAM.

Our FPGA also offers 8GB of on-board DRAM (which takes about 20 cycles between read request and response) and 10MB of eSRAM (which takes fixed 12 cycles between read request and response). Because of the multi-cycle latency for these two classes of memory, they are not suitable for storing data that must be read/written every cycle. Furthermore, both are more bandwidth-limited than BRAM and offer fewer lookups in parallel. However, as we will discuss in §4.2, pushing what data is feasible into these classes of memory is necessary to free up as much BRAM as possible to support fast-path memory-intensive processing.

2.3 FPGAs and IDS/IPS Performance

We are not the first to integrate FPGAs into IDS/IPS processing. However, prior work follows an 'offload' approach to utilizing the FPGA, where the CPU is 'primary' and performs the majority of processing, and the FPGA accelerates a single task [10, 16, 18, 19, 22, 34, 41–43, 46, 48, 49]. Most research in this space targets offloading regular expression matching alone [22, 41, 48], although some target TCP reassembly [42, 49] or header matching [27] instead. Unfortunately, a basic analysis based on Amdahl's law reveals why this approach fundamentally cannot bring IDS/IPS performance onto a single server at 100Gbps.

In Figure 2 we illustrate the *fraction of CPU time* spent on each task in Snort 3.0: *MSPM* (which, in Snort 3.0, implements header and partial string matching), *Full Matching* (which, in Snort, implements regular expressions and additional string matching), TCP *Reassembly*, and *other tasks*. As we can see, no single task dominates CPU time – at most, the MSPM consumes 46% of CPU time for one trace.

Using Amdahl's Law, we can see that even if MSPM were offloaded to an imaginary, *infinitely-fast* accelerator, throughput would increase by only 85% to 600Mbps/core, still requiring 166 cores to reach 100Gbps. In Figure 3, we show the idealized 'speedup factor' from offloading any individual module (assuming an infinite accelerator) for each of our traces; no module even reaches as much of a speedup as 2×.

The key lesson is simple: a much more drastic approach is needed to achieve line-rate throughput on a single server.

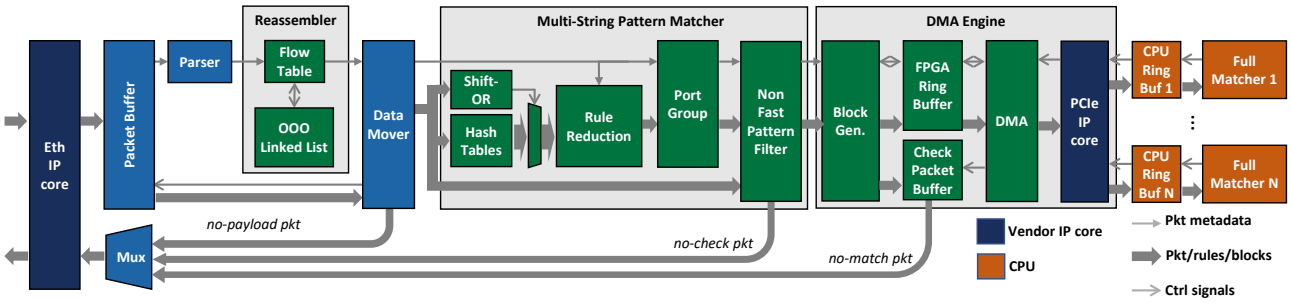


Figure 4: Pigasus architecture.

3 System Overview

We now present an overview of Pigasus. Recall that our target is to achieve 100Gbps, for 100K concurrent flows, and 10K rules within the footprint of a single server. We first describe our rationale behind Pigasus' *FPGA-first* approach (§3.1) before presenting Pigasus' packet processing datapath module by module (§3.2) and discussing how Pigasus makes best use of memory resources (§3.3). Finally, we lay out the threat model that we consider (§3.4).

3.1 An FPGA-First Design

Following our analysis in §2.3, we argue for an *FPGA-first* design for IDS/IPS processing. By FPGA-first, we mean that the FPGA is the *primary* compute platform – performing the majority of work – and that the CPU is *secondary*, operating only as needed. Following our analysis in §2.3, any approach to speed up IDS/IPSes by orders of magnitude must take on parallelizing *as much of the system as possible.*

We can even consider an extreme design, running the *entire system* on the FPGA, disposing the need for CPUs. However, we avoid this approach and choose instead to leave regular expressions and the 'full match' stage on traditional processors. The reason is simple – compared to the other packet processing modules, implementing the Full Match stage entirely on the FPGA provides lower performance benefits at a higher cost in terms of memory. As we will see in §5, the Full Match stage only interacts with ≈5% of packets in the Pigasus design. Hence, it is *not* a performance bottleneck for the majority of packets. Furthermore, regular expression parsing is a very mature research and yet state-of-the-art hardware algorithms do not reach our performance and memory demands for Pigasus. We estimate that GRAPEFRUIT [37], a state-of-the-art regular expression engine for FPGAs, would require 8MB of BRAM to statically map all the regular expressions from our ruleset on the FPGA, and yet would still only keep up with a few Gbps of traffic. Hence, we would likely need multiple replicas of the GRAPEFRUIT design – at least 24MB of BRAM – to keep up with the average of 5Gbps of traffic that reach the full matcher. Therefore, offloading regular expressions would exhaust our memory budget for little gain, in that *the majority of packets will never execute the full matcher anyway*.

3.2 Pigasus Datapath

Figure 4 depicts the major components of Pigasus' architecture. Notice that the Parser, Reassembler, and Multi-String Pattern Matcher (MSPM) are implemented in the FPGA while the Full Matcher is offloaded to the CPU.

Initial packet processing: Each packet first goes through a 100Gbps *Ethernet Core* that translates electric signals into raw Ethernet frames. These frames are temporarily stored in the *Packet Buffer*; each frame's header is separately sent to the *Parser* – which extracts TCP/IP header fields as metadata (*e.g.*, sequence numbers, ports) for use by the Reassembler and MSPM – and then forwards the header to the Reassembler.

Reassembler: The Reassembler sorts TCP packets in order so that they can be searched contiguously (*i.e.*, to identify matches that span across multiple packets). The Reassembler is able to record the last few bytes of the packet's predecessor in that flow in order to enable cross-packet search in the MSPM. UDP packets are forwarded through the Reassembler without processing. The key challenge in designing the Reassembler is doing so *at line rate* with state for *100K flows*; we discuss the design of the Reassembler in §4.

Data Mover: While the Parser and Reassembler operate on headers and metadata alone, the MSPM operates on full packet payloads. The Data Mover receives the (sorted) packet metadata from Reassembler and issues requests to fetch raw packets from the *Packet Buffer* so that they can be forwarded to the MSPM.

Multi-String Pattern Matcher: The MSPM is responsible for (a) checking every packet against the header match for all 10,000 rules, and (b) *every index of every packet* against all of the string-match filters for all 10,000 rules. Pigasus' MSPM does more work than Snort 3.0's equivalent module: Snort 3.0 searches for only *one* exact match string (called the fast pattern) in the MSPM, and pushes detection of the remainder of the strings to the 'full match' module. By searching for all of the exact match strings in the MSPM, Pigasus reduces the number of packets sent to the Full Match stage by more than 2× relative to Snort 3.0, and also reduces the number of suspected rules for the full matcher to check per packet by 4×. We describe the Pigasus MSPM design in §5.

DMA Engine: For each packet, the MSPM outputs the set of *rule IDs* that the packet partially matched. If the MSPM outputs the empty set, the packet is released to the network; otherwise it is forwarded to the DMA Engine which transfers the packet to the CPU for Full Matching. To save CPU cycles, the DMA Engine keeps a copy of the packets sent to the Full Matcher; this allows the Full Matcher to reply with a (packet ID, decision) tuple as a response rather than copying the entire packet back over PCIe after processing. The DMA Engine distributes tasks across cores in a round-robin fashion.

Full Matcher: On the software side, the Full Matcher polls a ring buffer which is populated by the DMA Engine. Each packet carries metadata including the *rule IDs* that the MSPM determined to be a partial match. For each rule ID, the Full Matcher retrieves the complete rule (including regular expressions) and checks for a *full match*. It then writes its decision (forward or drop) to a transmission ring buffer which is polled by the DMA Engine on the FPGA side. If the decision is to forward, the DMA Engine forwards the packet to the network; otherwise the packet is simply erased from the DMA Engine's Check Packet Buffer.

3.3 Memory Resource Management

The core obstacle to realizing an FPGA-first design is fitting all of the above functionality (except the full matcher) within the limited memory on the FPGA. As discussed in §2.2, BRAM is the 'best' of the available memory: it is the only class of memory that can perform read operations in one cycle, and it is also the most parallel. However, it is limited to only 16MB even on our high-end Intel Stratix 10 MX FPGA.

Therefore, we reserve BRAM only for modules which read or write to memory the most frequently, with multiple accesses per packet; namely the Reassembler and the Multi-String Pattern Matcher. Specifically, the Reassembler performs multiple accesses per packet as it needs to check for out-of-orderness, manage the out-of-order packet buffer, and check and release packet headers when an out-of-order 'hole' is filled. The MSPM too entails multiple memory accesses per packet, as every index of every packet must be checked against 10K exact-match string rules, and every packet header must be checked against the header matches for 10K rules.

To save BRAM, we allocate the other stateful modules such as the Packet Buffer and DMA Engine to less powerful eSRAM and DRAM respectively.[2] eSRAM and DRAM turn out to be sufficient for these tasks because the Packet Buffer and DMA Engine have much less stringent demands in terms of bandwidth and latency. In the case of the packet buffer, packet data is written and read only once and hence bandwidth demand is low but still exceeds DRAM's peak throughput; the data mover prefetches each packets 12 cycles before pushing it to the MSPM, keeping throughput high with a negligible latency overhead. The DMA Engine uses DRAM – which has the highest and variable latency and the lowest bandwidth – for the Check Packet Buffer. Since on average only 5% of packets require Full Matching functionality, this places little stress on DRAM bandwidth; the latency overhead of DRAM, while high when compared to BRAM, is still 10× faster than the PCIe latency suffered by packets sent to the CPU for full match.

Even though this leaves us with almost[3] the full capacity of BRAM for the Reassembler and Multi-String Pattern Matcher, realizing these modules is challenging. For instance, using traditional NFA-based search algorithms for the MSPM, given our public ruleset, would require 23MB – more than our 16MB BRAM capacity. Similarly, statically allocating 10KB of out-of-order buffer (*i.e.*, 10 packets) per flow for even 10K flows easily exceeds 100MB. Thus, in §4 and §5, we describe our design optimizations to ensure that the Reassembler and MSPM both 'fit' on-board without compromising performance.

3.4 Threat Model

Pigasus sits between an attacker and its intended target; an attacker may attempt to target Pigasus itself in order to indirectly damage the services Pigasus protects. We assume the attacker does not have physical access to the server running Pigasus, that the operating system and configuration tools for Pigasus are secure, and that the attacker cannot modify Pigasus' configuration. The attacker *may* inject arbitrary traffic into the input stream which Pigasus processes. In this context, we consider two classes of attacks. First, an attacker may attempt to *bypass* Pigasus – that is, send traffic which matches an IDS/IPS rule, but somehow 'trick' Pigasus into allowing it through – *e.g.*, by reording packets, sending duplicate packets, etc. Snort 3.0 employs numerous approaches to address these types of attacks (*e.g.*, timeouts and memory limits) which are straightforward for Pigasus to replicate. Second, an attacker may attempt to *slow down* Pigasus – sending out-of-order or very small packets to reduce Pigasus' effective throughput and hopefully stall or drop innocent traffic. These classes of attacks can always be overcome by 'scaling on demand' – *i.e.*, running more instances – but we set an additional goal of Pigasus being *at least as robust* as Snort 3.0.

4 Reassembly

Reassembly refers to the process of reconstructing a TCP bytestream in the presence of packet fragmentation, loss, and out-of-order delivery. Reassembly is necessary in Pigasus because the MSPM and Full Matcher must detect patterns (strings or regular expressions) that may span across more than one packet (*e.g.*, searching for the word 'attack' should

[2] FPGA manufacturers have been experimenting with varied classes of memory on-board the FPGA over the past few years. From the manufacturers' perspective, Pigasus can be seen as a success story for how varied memory enables more diverse applications which tailor their memory usage to per-task and data structure demands.

[3] We do use BRAM in some other places for internal buffers/queues.

not fail just because 'att' appears at the end of packet n and 'ack' appears at the beginning of packet $n+1$). Note that our goal is not a full TCP endpoint and hence we are not responsible, *e.g.*, for producing ACKs; the IDS/IPS is a passive observer of traffic between two existing endpoints, merely re-ordering the packets it observes for analysis. The key objective of our Reassembler is to perform this re-ordering for 100K's of flows, while operating at 100Gbps, *within the memory limitations* of our FPGA.

4.1 Design Space for TCP Reassembly

Hardware design often favors data structures that are *fixed-length*, *constant-time* and generally *deterministic*, and most TCP reassembly designs follow suit. For instance, [49] allocates a fixed 64KB packet buffer in DRAM and uses 7 pairs of pointers to track OOO state for each flow; similarly, [42] maps a fixed-sized 'segment array' in DRAM to track per-flow state. By using static buffers, these designs are guaranteed constant-time insertion of out-of-order packets into memory; furthermore, the memory consumed by any individual flow is fixed so freeing space is also deterministic. In addition, each flow is bounded in its resource consumption and so a highly out-of-order flow cannot take over the available address space, starving other flows.

The problem with these designs is that, by allocating a fixed buffer, they both *waste memory* and *limit out-of-order flows*. For example, allocating 64KB for *each and every flow* [42] would require 6.4GB to support 100K flows – orders of magnitude bigger than our BRAM capacity. Even worse, the vast majority of flows *don't need the space* most of the time because *most packets arrive in order*. On the other hand, flows which do suffer a burst of out-of-order packets (perhaps due to network loss) that exceeds the 64KB capacity cannot be served, even if there is memory available.

For software developers, the obvious response to these challenges is to use a more memory-dense data-structure such as a linked-list, where each arriving segment is allocated on-demand and inserted into the list in order. Because memory is allocated on demand, no memory is wasted, and those flows which need more capacity are able to consume more as available. In our empirical traces, 0.3% of packets arrive out of order, with 'holes' in the TCP bytestream typically filled in after 3 packet arrivals from the same flow. In a linked-list based design, this means that on average an out-of-order flow consumes 5K bytes at most.

From a hardware perspective, however, a linked list is an unorthodox choice: pipeline parallelism depends on each stage of the pipeline taking a fixed amount of time. Since linked lists have variable insertion times, depending upon how far into the list a segment must be inserted, linked lists can lead to pipeline stalls which result in non-work-conserving behavior upstream from the slow pipeline stage, and hence overall poor throughput. We find that by carefully designing the reassembly pipeline as a combination of a *fast path* (only handling in-order flows) and a *slow path* (that handles the remaining out-of-order flows), one can achieve the best of both worlds.

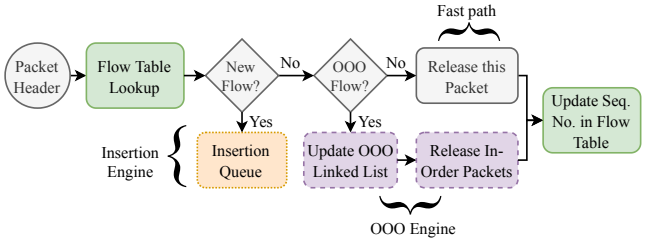


Figure 5: Reassembly Pipeline.

4.2 Pigasus TCP Reassembler

Pigasus takes the linked list approach, targeting a more memory efficient design. However, to avoid pipeline stalls due to variable-time packet insertions, Pigasus uses *three execution engines* to manage reassembly state, each of which handles a different class of incoming packet headers. The *Fast Path* processes in-order packets for established flows; the *Insertion Engine* handles SYN packets for new flows; and the *OOO Engine* handles out-of-order packets for existing flows. Because Pigasus is implemented in hardware, these engines can all run simultaneously (on different packet headers) without stalling each other, but must be careful not to conflict in accessing shared state in the Reassembly *Flow Table*. The flowchart in Figure 5 describes the sequence of steps that occur when a packet header arrives at the Reassembler.

The Flow Table, in representation, is a hash table mapping the classic flow 5-tuple identifier to a table containing (a) the *next expected sequence number* for an in-order packet, and (b) the *head* node for a linked list containing the headers of *out-of-order* packets waiting for a 'hole' in the TCP sequence number space to be filled. We discuss how the Flow Table is implemented in §4.3.

Fast Path: Upon arrival from the parser, each packet header is picked up by the Fast Path which looks up the flow's entry in the Flow Table. If no entry exists for that flow, the Fast Path pushes the packet header on to a queue for the Insertion Engine and moves on to the next packet. If there exists an entry for that flow, but (a) the packet header does not match the next expected sequence number in the Flow Table, or (b) the *head* node in the flow table is not null, the Fast Path pushes the packet header on to a queue for the OOO Engine. Finally, if the packet header *does* match the next expected sequence number in the flow, the Fast Path updates the expected sequence number in the Flow Table to the sequence number for the subsequent packet in the flow and pushes the current packet out towards the MSPM. Every task on the Fast Path runs in constant time, and so throughput is guaranteed through this engine to be 25 Million packets-per-second, which amounts to at least 100Gbps so long the average packet size is greater than 500B (Internet traces typically have an average packet

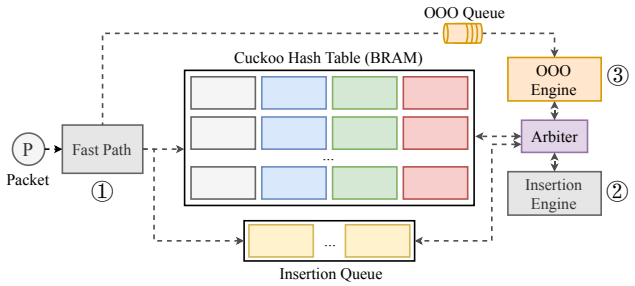


Figure 6: Flow Table and OOO Engine.

size of more than 800B [15]).

OOO Engine: The OOO Engine does not run in constant time, instead dequeueing packets provided for it from the Fast Path as it finishes operating over the previous packet.[4] For each dequeued packet, the OOO engine allocates a new node representing the packet's starting and ending sequence numbers, traverses the linked list for that flow, and inserts the newly-allocated node at the appropriate location. If the packet fills the first sequence number 'hole' in the linked list, then the OOO Engine removes the now-in-order packet headers from the list, releases them to the MSPM, and also updates the Flow Table entry with the new linked list head and next expected sequence number. If the OOO Engine detects that BRAM capacity for OOO flows exceeds 90% of its maximum capacity, it drops the flow with the longest linked list.

Insertion Engine: The Insertion Engine inserts new flow entries into the Flow Table; like the OOO Engine this path too can take variable time. We discuss the insertion engine in more detail in the next subsection.

Overall, allocating memory on-demand avoids memory wastage, and enables Pigasus to better serve OOO flows that do have a higher memory requirement. Additionally, bifurcating the reassembly pipeline into *fast* and *slow* paths prevents out-of-order flows – which require non-deterministic amounts of time to be served in our design – from impacting the performance of in-order flows, which represent the common case.

4.3 Implementing the Flow Table

While the Fast Path, Insertion Engine, and OOO Engine all operate simultaneously, they must synchronize over shared flow state (for instance, to keep the next expected sequence number for each flow consistent). We briefly discuss the implementation of our Flow Table that provides fast and safe concurrent access to these three engines.

The flow table design borrows a key data-structure from FlowBlaze [36]: an FPGA-based hash table that employs deamortized cuckoo hashing [8, 28]. We illustrate this data structure in Figure 6. The design provides high memory density (up to 97% occupancy using 4 or more sub-tables [8, 28, 36]), and worst-case constant-time reads, writes, and deletions for existing entries. It also guarantees that, for an Insertion Queue whose size is logarithmic in the number of flow table entries (in practice, a small value), the queue will not overflow [8]. We implement the hash table using dual-port BRAM, and the Insertion Queue using a parallel shift-register (capable of storing 8 elements).

The key to maintaining the hash table's deamortization property is the Insertion Engine, which is responsible for inserting: (a) new flows, and (b) flows that were previously evicted from the hash table during a 'cuckoo' step. Effectively, the Insertion Engine dequeues an element from the front of the Insertion Queue, and attempts to insert it into the hash table. If at least one of the 4 corresponding hash table entries is unoccupied, it simply updates the flow table and proceeds to the next queued element; otherwise, it *evicts* one of the 4 flow table entries at random, pushes the evicted entry onto the queue, and inserts the deqeued element in its place.

To guarantee conflict-free flow table access, we have the following prioritization of operations to the table. First, note that the Fast Path and OOO Engine never conflict over the same entry – the flow is either in order or not. The Insertion Engine *can* conflict with both the Fast Path and OOO Engine, as it may try to 'cuckoo' entries. Hence, we enforce the following priorities: (1) Fast Path > Insertion Engine (to ensure deterministic performance on the Fast Path), and (2) Insertion Engine > OOO Engine (to ensure that the queue drains and since, empirically, the OOO path is underutilized). Since our BRAM is dual-ported, we allow the Fast Path direct access to the Flow Table, while accesses originating from the OOO Engine or Insertion Engine are managed by an arbiter that enforces the aforementioned priority scheme.

4.4 Worst-Case Performance

Since Pigasus serves on the front-lines of network defenses, it is a prime target for Denial-of-Service (DoS) attacks. Most of Pigasus' underlying components are, by design, fully pipelined, enabling packet data to 'stream through' without ever stalling the system. However, this is not the case for the OOO path in the TCP reassembler. While all *in-order* packets are guaranteed full throughput, an attacker could potentially slow down the OOO path by injecting out-of-order flows into the system.

The key question is how this out of order traffic will impact 'normal' or 'innocent' TCP connections. We observe in our traces that 0.3% of packet arrivals from innocent connections arrive out of order, hence 99.7% of innocent traffic will 'stream through' the fast path, unimpacted by slowdowns on the OOO path. But, worst-case slowdowns on the OOO path *can* stall innocent traffic behind lengthy linked-list traversals due to a malicious sender.

Using mathematical models (elided for space), we quantify the performance of our system in terms of the *goodput*, or the

[4] Experimentally, we actually observe that this slow path is mostly idle when running over our traces, as most packets arrive in order or mostly in-order. We artificially stress this path to overload in our evaluation, but doing so requires an extreme rate of packet loss.

packet rate (in Gbps) corresponding to 'innocent' traffic that the system can sustain in steady state. Then, the attacker's objective is to inject adversarial traffic on the ingress link so as to minimize the achieved goodput.

Starting with a 100Gbps ingress link, we characterize the adversarial scenario using two parameters: the fraction of input traffic that is adversarial, a, and the fraction of non-adversarial input traffic that is in-order, t. Table 1 depicts the expected goodput for different values of a and t (using an average packet size of 500B for innocent traffic) according to the model. Ideally, all innocent traffic would traverse the system unhindered, but we see that slowdowns on the OOO path *can* make Pigasus fall short of this goal – especially at high rates of attack traffic injection – but that the OOO path is not entirely stalled and out of order packets do, eventually, make it through the system.

		In-order Traffic% (t)			
		99.7%	99.0%	90%	**Ideal**
Attack Traffic Rate (a)	10Mbps	99.7	99.1	91.5	99.99
	100Mbps	99.6	98.9	90.1	99.9
	1Gbps	98.7	98.0	89.1	99
	10Gbps	89.7	89.1	81.0	90

Table 1: Total Goodput (in Gbps) for various combinations of the attack traffic rate and fraction of in-order traffic. In-order traffic is isolated from slowdown, even when an adversary introduces substantial out-of-order flows.

5 Multi-String Pattern Matching

Checking tens of thousands of string patterns against a 100 Gbps bytestream makes the multi-string pattern matcher (MSPM) module by far the most operation-intensive and performance critical component in Pigasus.

Role of MSPM in IDS/IPS: As explained in Section 2, a Snort signature/rule comprises three classes of patterns: a *header match*, a set of *exact match strings*, and a set of *regular expressions*. A packet triggers the rule iff *all* patterns are identified.

To avoid checking every single pattern for every index and every packet, rulesets are designed for a two-step matching process. In Snort, the MSPM is responsible for checking *header matches* and *one, highly-selective exact match string*, called the *fast pattern*. Only packets which both match the header match and the fast pattern are forwarded to the *full matcher* which checks regular expressions and any secondary exact match strings (referred to as non-fast pattern strings). Pigasus' MSPM checks for fast patterns, headers, *and* non-fast patterns, reducing the load on the CPU-side full matcher.

MSPM Design Landscape: To the best of our knowledge, there are other no hardware or software projects reporting multi-string matching of tens thousands of strings at 100 Gbps. Classically implemented with parallel NFAs, the best hardware-based string matcher that we know of [16] would require 23MB of BRAM to represent the exact match search strings alone (ignoring the additional header matches).

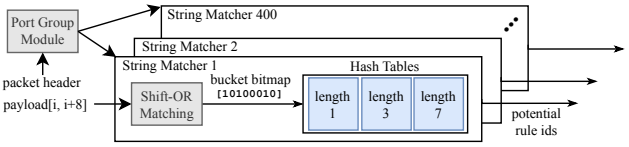


Figure 7: MSPM in Snort 3.0. Every String Matcher selected by the Port Group Module is evaluated sequentially.

To attain a more efficient design, we instead look to *software* and Intel's Hyperscan algorithm for string matching, which is AVX parallelizable and provided an 8× speedup compared to state-machine based string matchers in software [47]. Although naïvely re-implementing Hyperscan on the FPGA is in fact more memory intensive than the NFA approach (requiring 25MB to sustain 100Gbps), we find that by re-architecting Hyperscan's hash table-based design, we can reduce this memory footprint to only 2MB, leaving memory to spare and *expand* on the Hyperscan approach to search for *all* strings (rather than a fast-pattern only) for a total memory budget of 3.3MB. The key idea is to arrange hash table filters hierarchically, with *low memory* filters placed early in the pipeline with a *high replication factor*; this filters out a majority of traffic early. Subsequent stages of the MSPM may be more memory-intensive, per-module, but each stage handles less and less traffic and hence requires less replication.

In what follows, we first describe Hyperscan's two-stage MSPM, which checks for header matches and fast patterns. We then describe Pigasus' three-stage MSPM, which checks for fast patterns, header matches, and non-fast pattern strings using highly parallel, hierarchical filters to improve memory density.

5.1 MSPM in Software

Snort 3.0 + Hyperscan: In Snort 3.0, the MSPM is implemented using Intel's Hyperscan, illustrated in Figure 7.

Packets are first checked for their *header match*. Across all 10K rules, there are only ≈400 *unique* header match values. Rules which share the same header match fields are said to belong to the same *port group*. The port group module outputs a set of *port group IDs* which the packet matches; this output set is never empty because some rules wildcard their header match and hence match all packets. An average packet matches 2 port groups.

Packets are then checked for their *fast pattern string match*. For each port group, there exists a *string matcher* which checks fast patterns for all rules within that port group. Snort must check every string matcher for each port group the packet matches.

Within the string matcher, Snort must iterate over every index of the payload checking whether it matches any of the fast patterns in the port group. Rather than using a state machine

to do this, Hyperscan uses a collection of hash tables. For each possible fast pattern length[5] a hash table is instantiated containing the fast patterns of that length. Hyperscan then performs an exact-match lookup for all substrings at each index, looking up whether or not the substring is in the hash table – potentially $(8 \times l)$ lookups for a packet of length l.

To reduce the number of expensive sequential lookups, each string matcher contains a SIMD-optimized *shift-or filter* [9] prior to the hash table; this filter outputs either a '0' or '1' for every byte index of the packet, indicating whether or not that index matches *any* fast pattern in the hash tables; indices which result in a '0' output from the shift-or stage need not be checked.

The string matcher – combining shift-or and hash tables – then outputs a set of *rules* which the packet matched both in terms of *header* and *fast pattern*; together, the packet and the potential rule matches are passed to the full matcher. However, for 89% of packets, this stage outputs the empty set and the packet bypasses the full match stage entirely.

5.2 MSPM in Pigasus

A straightforward port of the Snort 3.0 MSPM engines and data structures onto the FPGA consumes 785KB of memory and forwards at a rate of 3.2Gbps. Taking advantage of the high degree of parallelism offered by the FPGA, one could, in theory, scale to 100Gbps via *data parallelism*, *i.e.*, replicating this 32 times. Unfortunately, doing so would require 25MB of BRAM. We now describe how Pigasus re-architects the Hyperscan algorithm to achieve this high degree of parallelism within available resources. Since this results in leftover memory, we can then *extend* Pigasus' MSPM to scan for non-fast pattern strings as well.

As shown in Figure 8, Pigasus flips the order of Hyperscan's MSPM, starting with string matching before moving on to header matching and port grouping.

Fast Pattern String Matching (FPSM): To perform string matching, Pigasus (like Hyperscan) also has a *filtering* stage in which packets traverse two parallel filters: a shift-or (borrowed from Hyperscan) and a set of per-fast-pattern-length hash tables. We check the shift-or and (32×8) hash tables in parallel. Hash tables only store 1-bit values indicating whether a given (index, length) tuple results in a match – but it does not store the 16-bit rule ID. The output from the filters is ANDed together, reducing *false positives* from either filter alone by 5×.

The shift-or and 1-bit hash table[6] consume only 65KB and 25KB respectively, thus they are relatively *cheap* to replicate 32× over in order to scale to 100Gbps. In theory, these filters can generate (32×8) matches per cycle (*i.e.*, 8 matches per filter); however, in the common case, most packets and most indices *do not match any rules*, and therefore require *no further processing*.[7] This gives us the opportunity to make subsequent pipeline stages narrower. We design a 'Rule Reduction' module that selects non-zero rule matches from the filter's 256-bit wide vector output and narrows it down to 8 values.

Applying this filter first allows us to use fewer replicas of subsequent data structures (which are larger and more expensive), since most bytestream indices have already been filtered out by the string matcher. This enables high (effective) parallelism with a lower memory overhead.

Header Matching: In this stage, we use the packet header data to determine whether the matches produced by the previous (FPSM) stage are consistent with the corresponding rule's Port Group. At this point, we only need to create 8 replicas of the 17KB Rule Table and 68KB Port Grouping modules to check 8 rules simultaneously.

Using the (index, length) tuple that resulted in a match in the FPSM stage, we look up the corresponding rule ID in the Rule Table. Next, using this rule ID, we look up the Port Group that this rule maps to; this could be a *single* port, a *list* or *range* of ports, or a *wildcard* (indicating a match on any port). If this packet's port number is a subset of this rule's Port Group, the rule is considered a match; otherwise, the rule is ignored.

Our initial design of the MSPM stopped here (at the Traffic Manager 1 stage in Figure 8), aiming merely to reproduce Snort's functionality, which only scans for fast patterns and headers. Packets which matched the fast pattern and header on at least one rule were sent to the CPU for processing. While packets which did not produce any matches at either the FPSM or Header Matching stage were simply streamed to the output interface.

This resulted in a design that sent 11% of packets to the CPU for processing, with an average of 4.4 rules searched per packet – and required only *2MB* of memory! Given that this amounted to a fraction of our resource budget for the MSPM, we asked ourselves: can we do more?

Non-Fast Pattern String Matching (NFPSM): Pigasus further filters down the packets and rules destined for the CPU to only 5% of packets, with just 1.1 rules/packet (on average), by additionally searching for *all* string matches within a rule on the board. Note that, on average, only 11% of packets reach the Non-Fast Pattern Matcher, and, by this point, we know which rules (on average 4.4 of them) the packet might match on. Naïvely, one might iteratively search for each string in the ≈ 4.4 rules, but because each packet has a variable number of rules and each rule has a variable number of strings (between 1 and 32), this approach would likely lead to low throughput and/or pipeline stalls.

Instead, Pigasus once again uses a set of hash tables (like in the FPSM) to search for all strings simultaneously. It then

[5] Up to 8 bytes – longer fast patterns are truncated.

[6] Subtly, this is not a true Bloom filter [13] because we only perform one hash per input; implementing multiple hashes increases resource utilization and complexity, we find, with little gain.

[7] Note that our filters never produce false negatives.

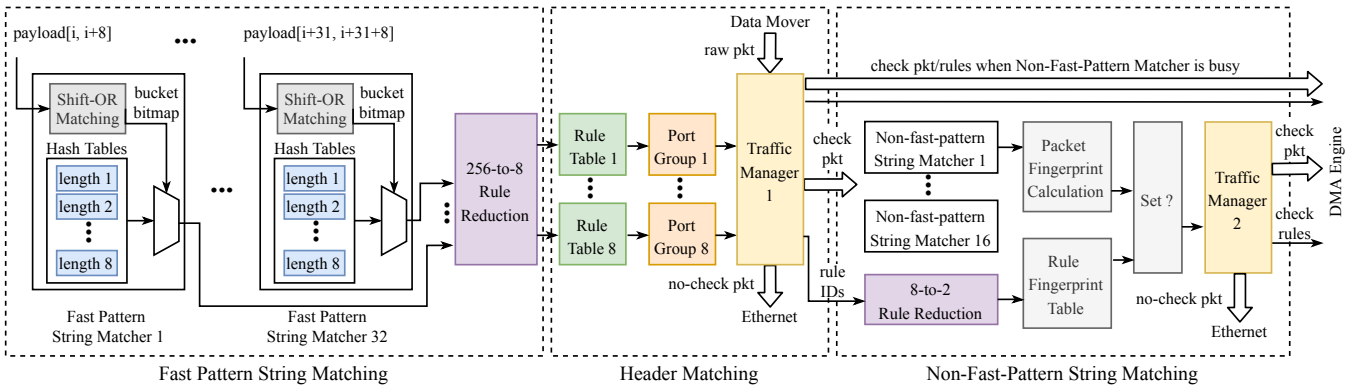


Figure 8: Pigasus' MSPM, which requires a total of 3.3MB of BRAM.

creates a compact, bloom-filter-like representation ('fingerprint') of the matched strings. To compute the fingerprint, we first represent the set of (index, length) tuples generated by the 8 NFPSM hash tables as a 16-bit vector by setting $bit[index \pmod{16}]$ to '1' for each length bucket. Next, for each bucket, all of the 16-bit vectors generated for a given packet are ORed together to create a 16-bit 'sub-fingerprint' for that bucket. Finally, these sub-fingerprints are concatenated into a 128-bit fingerprint representing the entire packet. The fingerprinting process is illustrated below:

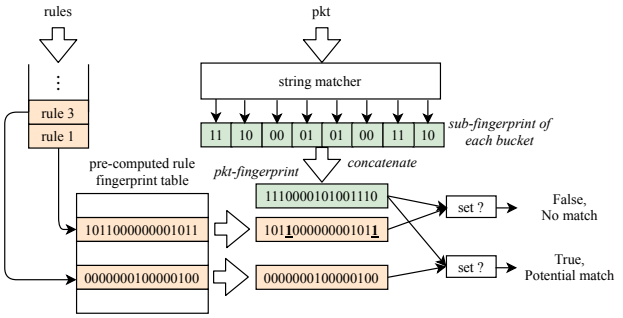


Figure 9: Rule matching fingerprints in the NFPSM.

The NFPSM can now look up a corresponding fingerprint – generated in the same way – for each of the ≈ 4.4 rules, and now can do a parallel set comparison between the two fingerprints. If, for every bit in the *rule's fingerprint*, the corresponding bit in the *packet's fingerprint* is also set, there is high probability that all of the exact match strings for the rule were matched. But, if any of the corresponding bits are *not* set, we can be certain that at least one of the non-fast pattern strings were not matched, thus eliminating the rule as a potential match. The 5% of packets which match at least one rule fingerprint are forwarded to the CPU; the remainder are released as non-matching and therefore innocent packets.

It is worth noting that, as the last stage of our hierarchical filtering, the non-fast pattern matcher has the lowest throughput capacity. This saves on resources, but can make the NFPSM vulnerable to overload. Where the fast pattern matcher is designed to process up to 100Gbps of incoming data, the non-fast pattern matcher tops out at a peak throughput of 50Gbps. 50Gbps is more than enough to handle the average rate of 11Gbps, but a spike of rule-matching malicious traffic can at times overload the non-fast pattern matcher. In this case, when the first traffic manager (between the header matcher and the non-fast pattern matcher) detects *backpressure* from the NFSM, it steers some packets directly to the CPU for checking. Temporarily increasing the load on software, where it is easier to 'scale out' and provision additional resources.

End-to-End: By hierarchically filtering out packets, the MSPM reduces the amount of traffic traversing each subsequent stage of the MSPM. This means that the earliest stages require high levels of replication, but the latter stages can, on average expect lower throughput and hence require less replication. Consequently, latter stages require lower memory consumption. End-to-end, the MSPM requires 3.3MB of memory, fitting well within our BRAM bounds while doing *more* filtering than what a naïve port of the Hyperscan algorithm would be capable of. Nonetheless, the reduced capacity of the MSPM in the latter phases of the MSPM does make these components vulnerable to overload; in these cases Pigasus temporarily shunts additional traffic to CPUs, where is easier to provision on demand and as needed.

6 Evaluation

In this section, we evaluate Pigasus and show that:

- Pigasus is at least an order of magnitude more efficient than state-of-art Snort running in software: using $23-200\times$ fewer cores, and $18-62\times$ less power;
- Pigasus' performance gains are resilient to a variety of factors such as small packets, out-of-order arrivals, and the rule-match profile of the traffic;
- The Pigasus architecture actually has resource headroom, suggesting a roadmap for handling even more complicated workloads.

We start by describing the evaluation setup we use for the rest of the section before the detailed results.

Module	ALM	BRAM (MB)	DSP	eSRAM (MB)
Packet Buffer	507 (0.1%)	0 (0%)	0 (0%)	5.91 (50.0%)
String Matcher	119,562 (17.0%)	3.30 (19.7%)	1,600 (40.4%)	0 (0%)
Flow Reassembler	20,728 (2.9%)	2.61 (15.6%)	0 (0%)	0 (0%)
DMA Engine	2,000 (0.3%)	0.32 (1.9%)	0 (0%)	0 (0%)
Instrumentation	1,189 (0.2%)	0 (0%)	0 (0%)	0 (0%)
Vendor IPs	42,028 (6.0%)	1.22 (7.3%)	0 (0%)	0 (0%)
Miscellaneous	21,946 (3.1%)	0.60 (3.6%)	0 (0%)	0 (0%)
Full Design	207,960 (29.6%)	8.05 (48.1%)	1,600 (40.4%)	5.91 (50.0%)

Table 2: Resource breakdown. Percentages are relative to the total amount of resources in a Stratix 10 MX FPGA.

6.1 Setup

Implementation and Resource Breakdown: We implement Pigasus using an Intel Stratix 10 MX FPGA Development card [2] as the SmartNIC in a 16-core (Intel i9-9960X @ 3.1 GHz) host machine. The Stratix 10 MX FPGA has 16MB of on-chip BRAM, 10MB of eSRAM, and 8GB of off-chip DRAM. Table 2 shows the FPGA resources used by each component of Pigasus when configuring it to support 100K flows and 10K rules. To implement Pigasus' CPU/software components, we adapt Snort 3 to allow it to receive reconstructed PDUs and rule IDs, coming from the FPGA directly into its Full Matcher. We run Snort 3 software experiments in an Intel i7-4790 CPU @ 3.60 GHz.

Traffic Generator: We installed both DPDK Pktgen [1] and Moongen [20] on a separate 4-core (Intel i7-4790 @ 3.6 GHz) machine with a 100Gbps Mellanox ConnectX-5 EN network adapter. DPDK Pktgen achieves higher throughput when replaying PCAP traces – up to 90Gbps – and hence we use the DPDK Packet Generator when running experiments with recorded traces. Moongen is better at generating synthetic traffic at runtime and can do so at up to the full 100Gbps offered by the underlying network. We specify in each experiment which traffic generator was used.

Traces and Ruleset: We test Snort and Pigasus both using the publicly available Snort Registered Ruleset (snapshot-29141) [6] and different traces from Stratosphere [44]: *CTU-Mixed-Capture-1–5*, *CTU-Normal-12*, and *CTU-Normal-7*. We refer to them as *mix-1–5*, *norm-1*, and *norm-2*, respectively. For the *mixed* traces, we use the `*before.infection` pcaps. We use Stratosphere traces because their packet captures contain the original payloads, which is essential when evaluating IDSes.

Measuring Throughput and Latency: We measure throughput in two ways: 1. The *Zero Loss* throughput is measured by gradually increasing the packet generator's transmission rate until the system (Snort or Pigasus) first starts dropping packets; 2. The *Average* throughput is computed as the ratio of the cumulative size of packets in the trace (in bits) to the total time required to process the trace. We measure latency (at low load) using DPDK Pktgen's built-in latency measurement routine. Unfortunately, DPDK embeds timestamps in the packet body, which never triggers the CPU-side Full Matching functionality. Instead, we measure the end-to-end latency for Pigasus on an empirical trace using FPGA-side counters, and then adding the baseline FPGA loopback latency to it.

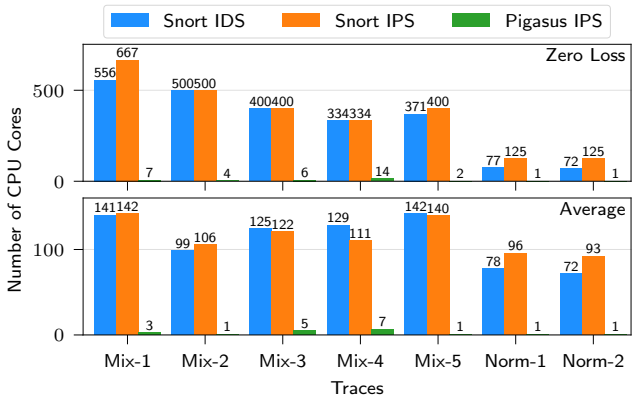


Figure 10: Number of cores required to process each trace at 100Gbps using Pigasus (FPGA + CPUs) and Snort (traditional CPUs alone). Pigasus numbers are based on implementation; Snort numbers are extrapolated from its single-core throughput and assume perfect linear scaling.

6.2 End-to-end performance and costs

In this section, we compare the performance, power, and cost of Pigasus vs. legacy Snort.

Provisioning for 100Gbps throughput: Figure 10 reports the number of server cores required to achieve 100Gbps for the evaluated Stratosphere traces for different settings. The top half is under the assumption of loss-free processing without buffering, while the bottom reports the steady-state core requirements based on the assumption that we could buffer packets during the peak periods and defer the full matching to allow the cores to catch up after the peak has passed.

The Pigasus results are based on experiments where the system is tested at increasing number of cores at maximum throughput, until we observe no packet loss. For the Snort experiments we run Snort in both IDS and IPS mode (with DPDK) on a single core and increase the throughput until it begins to drop packets. Note that while we report the actual number of cores required to run Pigasus, for Snort we extrapolate the single-core experiment to determine the number of cores that we would need to keep up with 100Gbps. This considers that Snort's throughput scales linearly with the number of cores and, therefore, represents an ideal *lower bound* to the actual number of cores needed to run Snort. Overall, we see that Snort in IDS mode requires $23-185\times$ more cores than Pigasus ($65\times$ on average), and in IPS mode requires $23-200\times$ more cores ($72\times$ on average).

Latency: Of course, in a practical IPS we care not only about throughput/provisioning but also per-packet latency. We plot the distribution of per-packet latency in Figure 11. We find

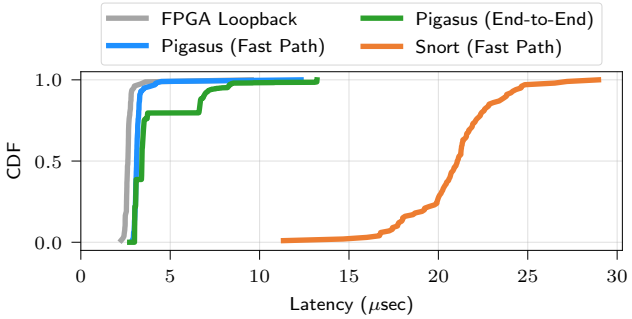


Figure 11: CDF of latency of Pigasus vs. Snort.

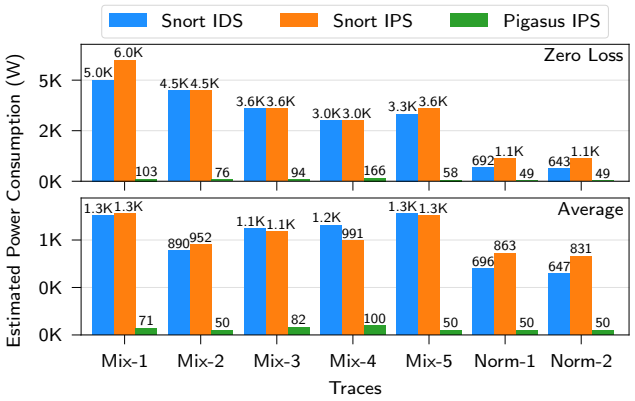


Figure 12: Estimated wattage to achieve 100Gbps.

that Pigasus yields almost an order of magnitude improvement in the median latency, and up to 3× improvement in the tail latency. As a point of comparison, we also show the baseline performance of a simple FPGA loopback measurement (*i.e.*, without any processing) and the Pigasus fast-path for packets that do not need further CPU processing. We find that the Pigasus fast path is very efficient and almost comparable to the baseline. We also find that Pigagus end-to-end latency only deviates substantially from the fast path for the tail. While we hypothesized some improvements in latency, we were puzzled by the magnitude of the improvement. Investigating why Snort was much slower revealed that on average, while Pigasus reduced the latency for the Reassembly (by 6μs), Parser (by 4μs), and the MSPM (by 3μs) as expected relative to software, the additional reduction came from avoiding Packet I/O overhead in software (around 5μs).

Power footprint: Figure 12 depicts the estimated power consumption required to achieve 100Gbps throughput for three configurations: Snort in IDS mode, Snort in IPS mode, and Pigasus in IPS mode. On the CPU side, we use Intel's Running Average Power Limit (RAPL) interface [23] to measure per-core power consumption in steady-state. To verify its accuracy we also measured the power utilization using an electricity usage monitor [4] and found consistent results. On the FPGA side, we use the Board Test System [2] (part of Intel's FPGA Development Kit) to measure power dissipation in the FPGA core and I/O shell. We observe that, across all traces, Snort (in *either* mode) has a $13-59\times$ higher power consumption than Pigasus (34× on average). We further note that the reported wattages for Pigasus represent a conservative estimate; while the total power consumption on the FPGA side is 40W, the core fabric accounts for just 13W, and the remainder is used for I/O (including Ethernet). Conversely, we only charge Snort for power consumed during compute tasks, ignoring other overages (such as Network I/O).

Cost: To estimate the Total Cost of Ownership (TCO), we consider both the capital investment and the power cost for each configuration. To estimate the capital investment, we use the per-core pricing data for the AMD EPYC 7452 CPU ($68.75 per core). For Pigasus, we also incorporate the market price of an Stratix 10 MX FPGA [2] ($10K). Assuming that the number of cores needed in practice is between the *Zero Loss* and *Average* in Figure 10, we estimate that the capital cost of the CPU-only solution is between $7,922 and $25,045, while the capital cost of Pigasus is between $10,189 and $10,344. To estimate the power costs we assume a lifetime of 3 years and electricity cost at $0.1334/kWh (average electricity rate in the US [3]). The power cost of the CPU-only solution at 9W/core is between $3,636 and $11,494, while the cost for Pigasus is between $227 and $298. Then, combining the capital investment and the power cost, the TCO of the CPU-only solution is between $11,558 and $36,539, while the TCO of Pigasus is between $10,416 and $10,642, saving between $1,142 and $25,897. We note that these estimates consider retail prices and do not account for other operational costs, such as cooling and rack space, which we expect to favor Pigasus. Moreover, for 100K flows and 10K rules we only use about half of a Stratix 10 MX; one may consider adapting the design to a smaller FPGA, further reducing the cost of Pigasus.

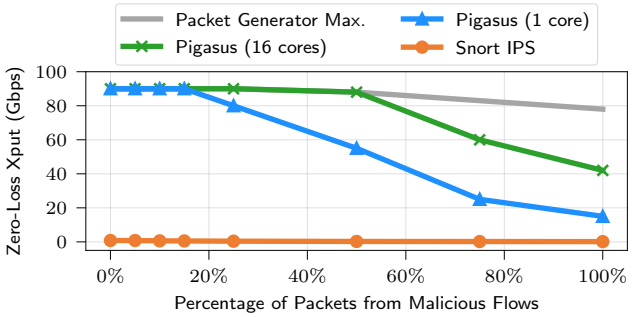
Recall that our original goal was to achieve 100Gbps supporting hundreds of thousands of flows matching tens of thousands of rules on a single server with a reasonable cost/resource footprint. The above results suggest that Pigasus indeed achieves this goal (with ample headroom).

6.3 Microbenchmarks and sensitivity analysis

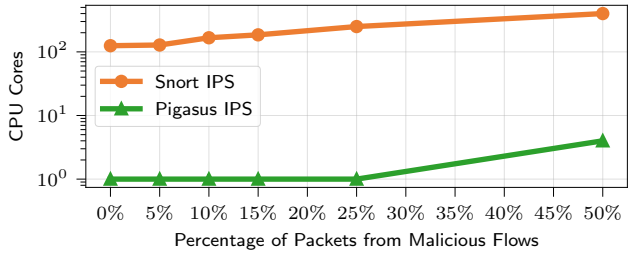
In this section, we present Pigasus' performance sensitivity to traffic characteristics. We probe deeper into Pigasus' performance under differing levels of malicious traffic. We further characterize the performance impact of *packet size*, and *out-of-order degree* of flows.

Dependence on CPU Offload: We construct semi-synthetic traffic traces by mixing malicious flows extracted from *mix-1* trace with innocent trace *norm-2* in different proportions.[8] Figure 13 (a) depicts the dependence of zero-loss throughput on the fraction of malicious flows (in terms of relative packet count). We report results for Pigasus (using both 1 and

[8] Note that not every packet in a malicious flow triggers a match.



(a) Zero-loss throughput.



(b) Number of cores required to achieve 100Gbps.

Figure 13: Impact of the fraction of malicious traffic on system throughput.

16 cores) and Snort IPS (with 1 core). We observe that, as long as the fraction of malicious traffic is smaller than 15%, Pigasus is able to process packets at line-rate using a single CPU core. With 16 cores Pigasus can process packets at line rate for up to 50% of malicious traffic. After the 50% mark, performance begins to degrade gradually. We repeated the same experiments disabling the software component of Pigasus and observed that the througput matches the 16-core experiment, suggesting that the hardware is the bottleneck. More specifically, the MSPM's rule reduction logic is stressed by the large number of potential rule matches.

Figure 13 (b) depicts the number of cores required to achieve 100Gbps as a function of the fraction of packets from malicious flows for up to 50%. Results for Snort are extrapolated from the single-core throughput. Despite the performance degradation observed in (a), Pigasus scales considerably better than Snort, requiring two orders of magnitude fewer cores. We also note that, while the hardware only becomes the bottleneck at an extreme fraction of malicious traffic, the design can be made even more robust using two hardware pipelines (discussed further in §6.4).

Dependence on Packet Size: We first consider the impact of packet size on Pigasus' performance stemming from the linked-list based TCP reassembler design. We configure the Moongen packet-generator to generate fixed-sized synthetic packets, and measure end-to-end, zero-loss throughput as we vary the packet size. Figure 14 illustrates this dependence. We observe that, for packets exceeding 500B (comparable to average packet sizes on the Internet [15]), Pigasus is capable of processing at line rate. (More generally, Pigasus by design

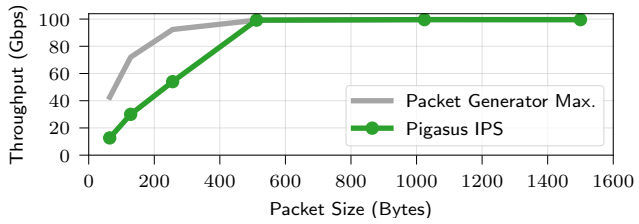


Figure 14: Zero-loss throughput achieved by Pigasus for a range of packet sizes.

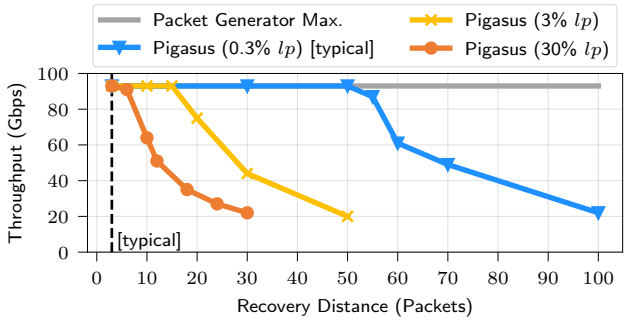


Figure 15: Zero-loss throughput achieved by Pigasus for a range of Loss Probabilities (lp) and Recovery Distances (rd).

can sustain 100Gbps as long as the average packet size is greater than 500B over a window of 87μs estimated base on buffer size.)

Dependence on Out of Order Degree: We characterize the OOO degree using randomly generated synthetic packet traces controlled by two independent variables: the packet loss probability (lp) [32] and the recovery distance (rd).[9] Figure 15 depicts the impact of these parameters on Pigasus' end-to-end, zero-loss throughput. We sweep the loss probability from 0.3% to 30%, and the recovery distance from 3 to 100. At typical values ($lp = 0.3\%$, $rd = 3$), Pigasus achieves a single-core throughput of 100Gbps, which degrades gradually with increasing packet loss and recovery distance. It is worthwhile to note that, at typical packet loss rates, the Reassembler can handle around 50 OOO packet arrivals without any degradation in end-to-end throughput.

6.4 Future outlook

Supporting 100Gbps with 100K flows and 10K rules requires only about half of the resources in our FPGA. We now explore what we can do with the additional capacity.

One option is to duplicate the existing processing pipeline (which runs at 100Gbps/25Mpps) each to serve a different subset of flows, increasing the throughput to 200Gbps, at the cost of creating additional copies of all the MSPM engines. Another option is to increase the number of supported flows

[9] Recovery distance is defined as the number of same-flow packets that arrive before a hole created by a lost packet is filled. In Pigasus, this value determines the amount of work (in cycles) that the OOO Engine must perform for each packet that arrives out of order.

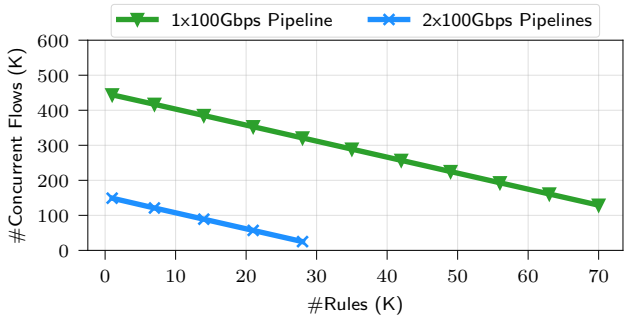


Figure 16: Tradeoff between the number of supported rules and concurrent flows when using one or two 100Gbps hardware pipelines.

or rules. Figure 16 depicts the three-way tradeoff between the scalability of the number of rules, concurrent flows, and replicated hardware pipelines. The design with two pipelines benefit from better throughput but have fewer room for storing rules or flows. There is plenty of scaling headroom in the Pigasus FPGA frontend design for more rules and flows.

7 Related Work

We now review some of the most related work, some of which served as inspiration for Pigasus.

Pattern matching: The design of the hash table filter in our Multi-String Pattern Matcher is similar to the filters used by DFC [17] and Hyperscan [47]. An important difference, however, is that instead of using a second hash table to associate potential string patterns with their identifiers, we directly use the matched index as a pattern identifier. This helps to reduce the amount of resources required by the hardware implementation. We also employ fewer, but much larger filters, since cache-friendliness is not a concern for FPGA design.

Using FPGAs to accelerate IDS/IPS: Many previous work have also made a case for using FPGAs to implement network functionality [21, 30, 33, 36, 45]. ClickNP [30] and FlowBlaze [36] present abstractions for making it easier to implement network functionality in FPGAs. However, they do not provide the necessary abstractions for searching bytestream nor they would be able to scale to meet our goals for throughput and number of rules. Some propose using FPGAs to accelerate IDS/IPS [10, 16, 18, 19, 34, 43, 46, 49]. However, all of these works do not implement a complete IDS/IPS and fail to meet our target for throughput or number of rules. Even though, Snort Offloader [43] proposes using an FPGA to implement an entire IDS/IPS, it only supports very simple operations, not including components that are essential for correct IPS operation, *e.g.*, TCP reassembly.

Other accelerators: Other works have looked at using hardware accelerators to improve some IDS/IPS components. Kargus [25] uses GPUs to accelerate exact-pattern and regular-expression matching. However, their use of GPUs contributes to increasing both power and latency. PPS [26] uses PISA switches to implement DFAs and accelerate arbitrary regular expressions. But are limited to only UDP and can only support a small number of string patterns. More important, however, we note that by only accelerating the latest IDS/IPS stages, these solutions are fundamentally limited in the throughput improvements they can achieve.

8 Conclusions

In many ways, IDS/IPS are one of the most stressful network workloads for both traditional software and hardware. As such, the gap between the workload demands and what was achievable on a single server always seemed elusive. The design of Pigasus is a singular proof point that a seemingly unattainable goal (100Gbps line rates for 100K+ flows matching 10K+ of complex rules) on a single server is well within our grasp. Looking forward, we believe that we can further unleash the potential benefits of FPGAs for this unique workload by further eliminating CPU bottlenecks and potentially moving additional functionality onto the FPGA. Given the future hardware roadmaps of FPGAs and SmartNICs, we believe that our insights and successes can more broadly inform in-network acceleration beyond IDS/IPS as well.

Acknowledgements

We thank the OSDI reviewers, Eriko Nurvitadhi, Aravind Dasu, and our shepherd Thomas Anderson and his students for comments and feedback on this work. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; in part by the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS); and finally in part by the project AIDA - Adaptive, Intelligent and Distributed Assurance Platform (reference POCI-01-0247-FEDER-045907), co-financed by the ERDF - European Regional Development Fund through the Operational Program for Competitiveness and Internationalisation - COMPETE 2020.

References

[1] DPDK-pktgen. https://github.com/Pktgen/Pktgen-DPDK.

[2] Intel Stratix 10 MX. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-mx.html.

[3] Microsoft Azure: Total cost of ownership (TCO) calculator. https://azure.microsoft.com/en-us/pricing/tco/calculator/. Accessed: 2020-10-01.

[4] PN1500 Watt meter. https://poniie.com/products/17.

[5] Snort 3. https://www.snort.org/snort3.

[6] Snort ruleset. https://www.snort.org/downloads/#rule-downloads.

[7] Zeek – network security monitor. https://www.zeek.org.

[8] Y. Arbitman, M. Naor, and G. Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Interna-*

tional Colloquium on Automata, Languages, and Programming, pages 107–118. Springer, 2009.

[9] R. Baeza-Yates and G. H. Gonnet. A New Approach to Text Searching. *Commun. ACM*, 35(10):74–82, Oct. 1992.

[10] Z. K. Baker and V. K. Prasanna. High-throughput linked-pattern matching for intrusion detection systems. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems*, ANCS '05, pages 193–202, New York, NY, USA, 2005. Association for Computing Machinery.

[11] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[12] BERTEN. GPU vs FPGA performance comparison. Technical Report BWP001, BERTEN Digital Signal Processing. http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf.

[13] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[14] M. Branscombe. The year of 100GbE in data center networks. https://www.datacenterknowledge.com/networks/year-100gbe-data-center-networks, Aug. 2018.

[15] CAIDA. Packet length distributions. https://www.caida.org/research/traffic-analysis/AIX/plen_hist/.

[16] M. Ceška, V. Havlena, L. Holík, J. Korenek, O. Lengál, D. Matoušek, J. Matoušek, J. Semric, and T. Vojnar. Deep packet inspection in FPGAs via approximate nondeterministic automata. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '19, pages 109–117, 2019.

[17] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han. DFC: Accelerating string pattern matching for network applications. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 551–565, Santa Clara, CA, Mar. 2016. USENIX Association.

[18] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '04, pages 249–257, 2004.

[19] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 24(1):52–61, Jan. 2004.

[20] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 275–287, New York, NY, USA, Oct. 2015. Association for Computing Machinery.

[21] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 51–66, Renton, WA, Apr. 2018. USENIX Association.

[22] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch. HARE: Hardware accelerator for regular expressions. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, Oct. 2016.

[23] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, Jan. 2012.

[24] Intel. FPGA design software – Intel Quartus Prime. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html. Accessed: 2020-10-14.

[25] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: A highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 317–328, New York, NY, USA, 2012. Association for Computing Machinery.

[26] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé. Fast string searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, pages 21–28, New York, NY, USA, 2019. Association for Computing Machinery.

[27] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 171–186, Renton, WA, Apr. 2018. USENIX Association.

[28] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2010.

[29] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. V. Lakshman. UNO: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519, New York, NY, USA, 2017. Association for Computing Machinery.

[30] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[31] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 363–378, Renton, WA, July 2019. USENIX Association.

[32] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997.

[33] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable router architecture for experimental research. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 1–7, New York, NY, USA, 2008. Association for Computing Machinery.

[34] P. Orosz, T. Tóthfalusi, and P. Varga. FPGA-assisted DPI systems: 100 Gbit/s and beyond. *IEEE Communications Surveys & Tutorials*, 21(2):2015–2040, 2019.

[35] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[36] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 531–548, Boston, MA, Feb. 2019. USENIX Association.

[37] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron. Grapefruit: An open-source, full-stack, and customizable automata processing on FPGAs. In *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '20, pages 138–147. IEEE, 2020.

[38] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, USA, 1999. USENIX Association.

[39] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 323–336, San Jose, CA, Apr. 2012. USENIX Association.

[40] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 13–24, New York, NY, USA, Aug. 2012. Association for Computing Machinery.

[41] R. Sidhu and V. Prasanna. Fast regular expression matching using fpgas. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, Los Alamitos, CA, USA, Apr. 2001. IEEE Computer Society.

[42] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley. Scalable 10gbps TCP/IP stack architecture for reconfigurable hardware. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '15, pages 36–43, USA, 2015. IEEE Computer Society.

[43] H. Song, T. Sproull, M. Attig, and J. Lockwood. Snort offloader: a reconfigurable hardware NIDS filter. In *International Conference on Field Programmable Logic and Applications*, FPL '05, pages 493–498, 2005.

[44] Stratosphere. Stratosphere laboratory datasets, 2015. Retrieved March 13, 2020, from https://www.stratosphereips.org/datasets-overview.

[45] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference*, ATC '17, pages 459–471, Santa Clara, CA, July 2017. USENIX Association.

[46] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, page 112–122, USA, 2005. IEEE Computer Society.

[47] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 631–648, Boston, MA, Feb. 2019. USENIX Association.

[48] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan. REAPR: Reconfigurable engine for automata processing. In *27th International Conference on Field Programmable Logic and Applications*, FPL '17, pages 1–8. IEEE, Sept. 2017.

[49] R. Yuan, Y. Weibing, C. Mingyu, Z. Xiaofang, and F. Jianping. Robust TCP reassembly with a hardware-based solution for backbone traffic. In *Proceedings of the 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*, NAS '10, pages 439–447, USA, 2010. IEEE Computer Society.

A Artifact Appendix

A.1 Abstract

Pigasus has a hardware component, that runs on an FPGA, and a software component which is adapted from Snort 3. The current version requires a host with a multi-core CPU and an Intel Stratix 10 MX FPGA (with 100 Gb Ethernet) [2]. Pigasus' artifacts are open source and publicly available.

We provide detailed instructions to reproduce Figure 10. This figure supports our main claim that Pigasus requires two-order of magnitude fewer cores than state-of-the-art Snort 3. In addition to the steps in this appendix and on the repository README, we also provide video archives that reproduce Figure 10 for both the Snort 3 Baseline[10] and the Pigasus[11] experiments.

A.2 Artifact check-list

- **Algorithm:** Pigasus Multi-String Pattern Matcher.
- **Program:** Snort 3 [5] for baseline experiments; DPDK pktgen [1] and Moongen [20] to generate packets.
- **Compilation:** Intel Quartus Prime [24].
- **Data set:** Stratosphere Laboratory Datasets [44].
- **Run-time environment:** System running Linux with Snort 3 [5] software dependencies installed. Quartus 19.3 with Stratix 10 device support is required to load the bitstream to the FPGA.
- **Hardware:** Two servers, one with an Intel Stratix 10 MX FPGA [2] and another with a DPDK-compatible 100 Gb NIC. Power-measurement experiments require either a CPU with a power measurement interface (*e.g.*, RAPL [23]) or an external electricity usage monitor.
- **Execution:** Disable power optimizations in the BIOS, isolate cores from the Linux scheduler, and pin processes to cores.
- **Experiments:** Experiments are run manually with Pigasus on one machine and a packet generator on another.
- **Public link:** https://github.com/cmu-snap/pigasus
- **Code licenses:** 'BSD 3-Clause Clear License' for the hardware component and 'GNU General Public License v2.0' for the software component. Check the repository for details.

A.3 Description

How to access

To access the artifact, clone the repository from GitHub:

```
$ git clone https://github.com/cmu-snap/pigasus.git
```

This repository also includes a README with the most up-to-date instructions on how to install and extend Pigasus.

[10] https://figshare.com/articles/media/snort_baseline_mp4/12922160

[11] https://figshare.com/articles/media/pigasus/12922178

Hardware dependencies

Pigasus requires a host with an Intel Stratix 10 MX FPGA [2]. This host should have PCIe Gen3 or greater and a slot with 16 lanes for the FPGA. Experiments require an extra host equipped with a DPDK-compatible 100 Gb NIC to be used as a packet generator. For the experiments, the two hosts are connected back to back. The power-measurement experiments require either a CPU with a power measurement interface (*e.g.*, RAPL [23]) or the use of an external electricity usage monitor.

Software dependencies

Pigasus' software component is adapted from Snort 3 [5] and inherits the same software dependencies. §A.4 provides instructions on how to install those. The provided implementation works on Linux only and was tested on Ubuntu 16.04 and 18.04. Experiments require the installation of vanilla Snort 3, for comparison, as well as DPDK pktgen and Moongen in the packet generator host. To be able to load the bitstream on the FPGA, an installation of Quartus 19.3 as well as the Stratix 10 device support are required.[12]

Data sets

To obtain the Stratosphere traces go to https://www.stratosphereips.org/datasets-overview.

A.4 Installation

These instructions assume that you already have the bitstream to be loaded on the FPGA. For instructions on how to synthesize the design, refer to the repository README.

Software Configuration

In a system running a fresh install of Ubuntu 18.04, with the Pigasus repository cloned to the home directory, start by setting the required environment variables and useful aliases by adding the following to your `.bashrc` or equivalent:

```
export pigasus_rep_dir=$HOME/pigasus
export pigasus_inst=$HOME/pigasus_install
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
export LUA_PATH="$pigasus_inst/include/snort/lua/?.lua;;"

alias pigasus="taskset --cpu-list 0 $pigasus_inst/bin/snort"
alias sudo='sudo '
```

Make sure you apply these changes:

```
$ source ~/.bashrc
```

Then install the dependencies using the provided script:

```
$ cd $pigasus_rep_dir
$ ./install_deps.sh
```

Once the dependencies are installed, build Pigasus as follows:

```
$ cd $pigasus_rep_dir/software
$ ./configure_cmake.sh --prefix=$pigasus_inst
     --enable-pigasus --enable-tsc-clock
     --builddir=build_pigasus
$ cd build_pigasus
$ make -j $(nproc) install
```

[12] Both can be obtained at: https://fpgasoftware.intel.com/19.3/.

Hardware Configuration

To load the bitstream make sure the Quartus tools are in your path by setting the following environment variables in your `.bashrc` or equivalent:

```
# quartus_dir should point to the Quartus installation dir.
export quartus_dir=
export INTELFPGAOCLSDKROOT="$quartus_dir/19.3/hld"
export QUARTUS_ROOTDIR="$quartus_dir/19.3/quartus"
export QSYS_ROOTDIR="$quartus_dir/19.3/qsys/bin"
export IP_ROOTDIR="$quartus_dir/19.3/ip/"
export PATH=$quartus_dir/19.3/quartus/bin:$PATH
export PATH=$quartus_dir/19.3/modelsim_ase/linuxaloem:$PATH
export PATH=$quartus_dir/19.3/quartus/sopc_builder/bin:$PATH
```

Make sure you apply these changes:

```
$ source ~/.bashrc
```

A.5 Evaluation and expected result

In what follows, we describe how to run the experiments to reproduce Pigasus results from Figure 10. Before every experiment we reload the bitstream on the FPGA and reboot the server. This ensures that we always start from the same FPGA state:

```
$ cd $pigasus_rep_dir/pigasus/hardware/hw_test/
$ ./load_bitstream.sh
$ sudo reboot
```

Once the machine is back, to run the software component, first insert the kernel module:

```
$ cd $pigasus_rep_dir/software/src/pigasus/pcie/kernel/linux
$ sudo ./install
```

Then, run Pigasus, using the following command:

```
$ cd $pigasus_rep_dir/software/lua
$ sudo pigasus -c snort.lua --patterns ~/rule_list
```

The `snort.lua` uses the same syntax as in Snort 3, you should modify it to include the Snort Registered Rule Set [6]. In our experiments, we modified the rules to remove some features currently not supported by Pigasus, including `services`, `file_data` and `nocase`. We also use the same modified rules in the baseline experiment.

When Pigasus finishes the startup process it will stop printing logs to the screen. Once this happens, you can invoke the FPGA JTAG console to configure the FPGA internal state. To do so, open another terminal and enter:

```
$ cd $pigasus_rep_dir/hardware/hw_test/
$ ./run_console
% source path.tcl
```

If the last command produces an error, exit the JTAG console with `Ctrl+C` and rerun the last two commands. Once the last command runs successfully type the following commands to configure the buffer size, set the number of cores, and check the FPGA internal state:

```
% set_buf_size 262143
% set_core_num 1
% get_results
```

This last command should return all zeros as no packets have been sent yet.

Now that Pigasus is running and properly configured, we can start the packet generator on another machine. Here we assume that DPDK pktgen is properly configured on the other machine and has been started.

You can specify the rate to send packets, where 100 means 100% line rate. To ensure that DPDK pktgen will only send the trace once, specify the number of packets to match the trace size. The example pcap we are using is the `norm-2.pcap`, which has 456,709 packets. After setting these parameters, you can start sending packets.

```
Pktgen:/> set 0 count 456709
Pktgen:/> set 0 rate 100
Pktgen:/> str
```

Once the packet generator finishes sending packets, go back to the JTAG console on the other host and type the following:

```
% get_results
```

This should return 456,709 received packets and 456,709 processing packets. This means that Pigasus processed all the packets sent at max rate, without loss.

Now stop Pigasus by going back to the first terminal and typing `Ctrl+C`. It will print `rx_pkt`, which should match the `dma_pkt` reported by the FPGA in second terminal. This means that all packets sent from the FPGA to the CPU for full evaluation were processed.

A.6 Experiment customization

Experiments may be customized to use different rule sets and different packet traces. Pigasus design can also be changed to support a different number of concurrent flows or rules.

A.7 Artifact Evaluation Methodology

Submission, reviewing and badging methodology: https://www.usenix.org/conference/osdi20/call-for-artifacts



DORY: An Encrypted Search System with Distributed Trust

Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica

University of California, Berkeley

Abstract. Efficient, leakage-free search on encrypted data has remained an unsolved problem for the last two decades; efficient schemes are vulnerable to leakage-abuse attacks, and schemes that eliminate leakage are impractical to deploy. To overcome this tradeoff, we reexamine the system model. We surveyed five companies providing end-to-end encrypted filesharing to better understand what they require from an encrypted search system. Based on our findings, we design and build DORY, an encrypted search system that addresses real-world requirements and protects search access patterns; namely, when a user searches for a keyword over the files within a folder, the server learns only that a search happens in that folder, but does not learn which documents match the search, the number of documents that match, or other information about the keyword. DORY splits trust between multiple servers to protect against a malicious attacker who controls all but one of the servers. We develop new cryptographic and systems techniques to meet the efficiency and trust model requirements outlined by the companies we surveyed. We implement DORY and show that it performs orders of magnitude better than a baseline built on ORAM. Parallelized across 8 servers, each with 16 CPUs, DORY takes 116ms to search roughly 50K documents and 862ms to search over 1M documents.

1 Introduction

Users have grown increasingly reliant on filesharing systems such as Box, Dropbox, and iCloud. However, attacks on storage servers [88, 95, 98, 109] have exfiltrated large amounts of sensitive data belonging to many users, jeopardizing user privacy as well as the reputation and business of the victim organizations. End-to-end encrypted storage systems [73, 107, 115, 121, 124] provide a strong defense against this type of attack: the client stores all cryptographic keys and the server receives only encrypted data, and so an attacker that compromises the server can only exfiltrate encrypted data.

At the same time, end-to-end encrypted filesharing services struggle to provide the same functionality as plaintext storage providers like Dropbox because the server cannot decrypt the data to process it. Server-side search is a critical tool that users expect for convenience and companies require for compliance.

Despite a large body of work on searchable encryption [23, 25, 35, 37–40, 50, 52, 67, 68, 94, 97, 111, 114, 116], *practical and leakage-free* search on encrypted data has remained an unsolved problem for two decades. Existing work can largely be divided in two categories: (1) practical but leaking search access patterns, or (2) not leaking search access patterns but expensive.

In the first category, an attacker can learn sensitive data by observing search access patterns. We now explain what search access patterns are intuitively by contrasting them to the leakage already existing in deployed end-to-end encrypted filesystems [73, 107, 115, 121, 124]. In these filesystems, when a user accesses a file, the server learns that this specific user accessed that specific file, but it does not see the content due to end-to-end encryption. The concern with leaking search access patterns on top of this filesystem leakage is that search access patterns can leak information at the word level, allowing an attacker to potentially reconstruct search queries and document plaintext [22, 65, 72, 84, 102, 106, 129].

Consider a simple example of how an attacker can exploit search access patterns [129]. The server stores an inverted search index for Alice's emails mapping an encrypted keyword to an encrypted list of files. The attacker sends a one-word email to Alice containing "flu". If Alice's client updates entry 924 of index on the server, the attacker learns that index[924] is for "flu". By repeating this process for every word in the dictionary, the attacker can discover the word corresponding to every index entry. Later when Alice receives a confidential email, the attacker can derive all the words in that email based on which index entries are updated. More sophisticated attacks can reconstruct both entire documents and search queries from even more advanced search schemes [22, 65, 72, 84, 102, 106, 129]. In this paper, we informally define search access pattern leakage as the set of documents matching a search keyword, the size of that set, and any information about the search query. In contrast, if a scheme does not leak search access patterns, then during a search on a folder, the search server learns only that a search is now happening in that folder.

The second category of existing work typically relies on Oblivious RAM (ORAM) [54, 99, 119], a cryptographic tool that allows a client to read and write data from a server without revealing access patterns. Many academic works point to an inverted index inside ORAM as a straightforward way to eliminate leakage [61, 96, 116]. Unfortunately, even though the asymptotic complexity of ORAM is polylogarithmic in the index size, the cost of even the most practical ORAM schemes remains prohibitively expensive for our setting. For example, inserting a file requires an expensive ORAM operation for *every* keyword in that file (and there can be hundreds).

Given that practical, leakage-free search remains a difficult

problem, we revisit the system model: What do real end-to-end encrypted filesharing systems actually require from a search system? Would the problem become more tractable in their system model?

Choosing a system model. We surveyed five companies that provide end-to-end encrypted filesharing, email, and/or chat services: Keybase [73], PreVeil [107], SpiderOak [115], Sync [121], and Tresorit [124]. To the best of our knowledge, this is the first study of requirements for encrypted search in real filesharing systems. We discuss our findings in §2 and summarize the ones most relevant to DORY here:

Efficiency requirements. These companies care about two primary metrics: latency and monetary cost. They are not concerned about the asymptotic complexity of the search algorithm and would accept an algorithm with runtime linear in the number of documents as long as their concrete performance and cost requirements are met (see Table 2).

Trust model requirements. Some of these companies were already splitting trust to back up secret keys or distribute public keys, and we wanted to know if we could leverage a similar distributed trust assumption to make the problem of encrypted search more tractable. While these companies were willing to split trust across multiple domains, some had two requirements aimed at strengthening the distributed trust assumptions. First, if at least one trust domain is honest, then an attacker that controls all the remaining trust domains and observes user queries should not learn search access patterns. In particular, we need to protect against a *malicious* attacker rather than an honest-but-curious one and should not assume that the attacker follows the protocol. The second requirement, stated intuitively, is that only search access patterns should be protected by distributed trust, and an attacker that compromises *all* trust domains should not immediately learn the contents of the search index.

While prior work explores some forms of distributing trust for encrypted search [15, 19, 45, 62, 64, 108], we are not aware of any work that meets both the efficiency and distributed trust requirements outlined above without leaking any search access patterns, as explained in §8.

Our system: DORY. We design and implement DORY (Decentralized Oblivious Retrieval sYstem), an encrypted-search system that splits trust to meet the real-world efficiency and trust requirements summarized above (and detailed in §2). DORY ensures that an attacker who cannot compromise every trust domain does not learn search access patterns.

We implemented and evaluated DORY to show that it performs better (for some metrics, orders of magnitude better) than an ORAM baseline (§7). DORY also meets the companies' efficiency requirements; parallelized across 8 servers, searching over 1M documents takes 862ms, and, using workload estimates from the companies, we estimate that DORY costs roughly $0.0509 per user per month.

DORY combines cryptographic and systems techniques to overcome the security and efficiency challenges of previous solutions. Several of the companies we surveyed have expressed interest in deploying DORY, and one of them already has plans to integrate DORY into their system in the near future.

1.1 Summary of techniques

Choosing an oblivious primitive. Given the inefficiencies of ORAM, a key challenge was choosing a cryptographic primitive for hiding search access patterns. We identified a relatively recent cryptographic tool, distributed point functions (DPFs) [51] (a specific type of function secret sharing [20, 21]), as particularly promising for our setting. DPFs allow us to leverage ℓ servers (for practical constructions, $\ell = 2$) to retrieve part of the search index without any group of $< \ell$ servers learning which part of the index we're retrieving (the problem of private information retrieval, or PIR [27, 28]). A DPF-based solution requires a linear scan over the index, but the overhead per index entry is small because it relies on AES evaluations, which are implemented efficiently in hardware.

Designing the search index. An important challenge is how to structure the search index to support efficient search and update operations. To minimize the overhead of updating the search index when a file is uploaded, the client should only need to upload a small, constant-sized amount of data per file, and ideally avoid performing an expensive cryptographic operation for every keyword in that file. To minimize search overhead, we need to limit the number of DPF queries. To achieve both of these goals, we keep a table where each row corresponds to a bitmap of words for a document. An update simply requires the client to insert a *row* by uploading a new bitmap, and, a search only requires a single DPF request to retrieve the *column* corresponding to a keyword (§4.1). However, this bitmap can become quite large to accommodate every word in the dictionary. To reduce the size of this bitmap (and thus the time for the linear scan), we use a Bloom filter, which provides compression while preserving column alignment. Bandwidth from the servers to the client is linear in the number of files searched over, but we require less than 1 byte per file (§7) and, more importantly, this fixed bandwidth enables DORY to hide the number of search results, which can be exploited in volume-based attacks [22, 72, 102].

Encrypting the search index. To prevent an attacker that compromises all the servers from immediately reconstructing the plaintext search index, we need to encrypt each bit in the Bloom filter before inserting it into the search index. Unfortunately, the expansion of encryption would increase the size of the search index (and thus the time for the linear scan) by the security parameter λ (typically $\lambda = 128$). To ensure that the encrypted index is the same size as the plaintext index, we instead mask the bits using a random one-time pad that we ensure is unique for each version of the file (§4.1).

Defending against a malicious attacker. DPFs do not protect against malicious attackers. To protect against a malicious attacker that compromises all but one of the trust domains, we leverage MACs to allow the client to check the integrity

of search results in a way that makes blackbox use of DPFs. Applied naively, adding MACs would increase the search bandwidth and storage at the server by a factor of λ. To address this problem, we employ aggregate MACs [71] to turn λ from a multiplicative factor to an additive one (§4.3).

Providing fault tolerance. Splitting trust across different trust domains naturally requires additional servers. With secret-sharing, one tool for distributing trust, servers store different data that they may not share. Then, to provide fault tolerance, each of these servers would need to be replicated. We observe that in DORY, servers can use each other for fault-tolerance even though they are in *different* trust domains due to two properties (§5): (1) each server has an identical copy of the state, and (2) the client can perform integrity checks.

Reducing the cost of replication. To execute a search query correctly, all the servers must operate on the same version of the state. This is challenging because clients can issue update and search requests concurrently. One possibility is to use standard Byzantine fault-tolerant (BFT) consensus techniques to solve this problem, but this would require $3f+1$ trust domains to handle f failures. Instead, we observe (1) the ways in which our system setting is less demanding than that of BFT, and (2) that our cryptographic protocol enables clients to check integrity even if all servers are compromised; using these, DORY only needs $f+1$ trust domains (§5).

2 Finding DORY: identifying a system model

To understand real-world use cases, we surveyed five companies providing end-to-end encrypted file storage, email, and/or chat solutions: Keybase [73], PreVeil [107], SpiderOak [115], Sync [121], and Tresorit [124]. For each company, we asked a set of questions (see full version [36]) over the course of discussion(s) and email exchanges. This study was conducted as we were in the process of designing our system. We summarize our findings in Tables 1 and 2 and in the following sections. We report statistics in aggregate to preserve the confidentiality of individual companies, as they requested. These statistics and requirements motivate DORY's system model.

About the companies. Before we report the results of our survey, we give a brief background about each company. Keybase [73], founded in 2014 in the US and recently acquired by the video-conferencing company Zoom [128], keeps a publicly auditable key directory and offers open-source, end-to-end encrypted chat and storage systems. PreVeil [107], founded in 2015 in the US, focuses on both encrypted chat and storage solutions and open-sources some of its tools. SpiderOak [115], founded in 2007 in the US, offers encrypted storage, backup, and messaging solutions leveraging a private blockchain and open-sources many of its tools. Sync [121], founded in 2011 in Canada, and Tresorit [124], founded in 2011 in Switzerland, both provide encrypted storage. With the exception of Keybase, these companies generally target enterprise customers and support compliance with regulations

	Keybase	PreVeil	SpiderOak	Sync	Tresorit
Need server search?	✓	✓	✓	✓	✓
Have server search?	✗	✗	✗	✗	✗
File sharing?	✓	✓	✓	✓	✓
Email?	✗	✓	✗	✗	✗
Chat?	✓	✗	✓	✗	✗
Mobile client?	✓	✓	✓	✗	✓

Table 1: The search use-cases for each of the five companies we surveyed.

Table 2: Survey statistics. In accordance with the companies' confidentiality wishes, we report most fields in aggregate although we report individual responses for max permissible search latency (only 4 of the companies responded).

System cost & scale	
Avg. #docs/user	100 - 45K
Max #docs/user	100K - 1.3M
Price/month/user	$0-20
Search requirements	
Max added $/month/user	$0.70-5.54
Max search latencies (s)	[0.5, 1, 1, 4]
Est. update/search ratio	50/50

such as GDPR or CMMC. Some of these companies report over 750K users in over 180 countries.

The need for server-side search. Every company expressed a need for server-side search on encrypted data either for their desktop client in cases where users do not have all the files downloaded, or for the mobile or web clients. However, none currently support server-side search; they all told us that they tried at some point to develop a solution (most had researched the academic literature), but their efforts were eventually thwarted by concerns about performance or search access patterns. Several of the companies we surveyed had built or used a client index as a temporary solution, but they did not see this as a long-term solution because of its inability to index many files locally (e.g. enterprise data) or its resource consumption (especially on mobile). In §7.5, we discuss how synchronization between clients makes this solution infeasible in cases where documents are constantly updated.

They all stated interest in deploying a server-side solution that met their functionality, security, and performance requirements, if such a solution were to exist.

2.1 System requirements

Search must be responsive. The companies reported maximum search latencies between 500ms and 4s (Table 2). The company that reported a maximum search latency of 500ms reported tens of thousands to hundreds of thousands documents per user, while some of the companies that reported larger maximum search latencies had users with approximately a million documents.

Monetary cost for search must be small. These companies prioritize keeping the cost of search below $0.70 per user per month in order to make it feasible to deploy search to all users without increasing prices (Table 2). While some companies were willing to consider charging more for the ability to search, other companies believed that users would be unwilling to pay

extra because they are used to free search on other platforms.

Multiple users must be able to update and search the same documents. Each company allows multiple users to access the same file. Therefore, a search solution should be designed with multiple clients in mind and minimize the amount of state clients need to synchronize between operations.

Revoking a user's access must be cheap. All these companies implement revocation lazily [9, 48, 53, 56, 66, 110], meaning that when a user's access to a folder is revoked, the remaining users generate a new key and, rather than re-encrypting every document in the folder under the new key, simply use the new key for subsequent updates. In this way, the revoked user can still access documents that haven't been updated since the time of revocation. These companies want to adopt a similar approach for search. When a user is revoked, rather than re-computing the entire search index (as in ORAM-based solutions), subsequent updates should not allow the revoked user to search over the updated documents.

Relaxations. In addition to learning requirements, we also learned several system relaxations these companies accepted. The companies did not require search results to be fresh (they could be stale for up to a few minutes), and they were also willing to accept a small number of false positives (several other search schemes have also leveraged this allowance [15, 52]).

2.2 Distributed trust requirements

The majority of prior encrypted search work considers a single-server model where the attacker can take control of the entire system. As some of these companies were already leveraging distributed trust (e.g. Keybase to distribute public keys via social media servers, PreVeil to backup secret keys secret-shared among multiple clients), we wanted to know if they were willing to accept a distributed trust model for encrypted search as well, as this could be an opportunity for providing a more efficient search. We found that all the companies were open to a distributed trust model, although several companies had more specific requirements for how to distribute trust:

Hide search access patterns even with only one honest trust domain. These companies wanted the guarantee that if at least one trust domain is honest, then an attacker cannot learn search access patterns. They did not want to assume that other trust domains behaved correctly, so they wanted a malicious threat model rather than an honest-but-curious one.

Distributed trust only for search access patterns. These companies wanted to limit the damage caused by an attacker who compromises *all* the ℓ trust domains by ensuring that putting the ℓ search indices together does not readily provide the attacker with the plaintext search index. For example, if a company is subpoenaed and every trust domain must hand over its search index and search access patterns from then on, the company can choose to suspend search services to protect users' privacy by reducing search access pattern leakage, similar to the case where Lavabit chose to suspend operation rather than reveal Snowden's emails [4]. In such a case, reconstructing the index from the ℓ servers' index shares should result in end-to-end encrypted data. This requirement rules out solutions based on secret-sharing a plaintext search index across multiple servers because an attacker compromising all trust domains can recover the plaintext index.

2.3 Opportunities

From the survey results reported above, we summarize what we considered opportunities to make the problem of encrypted search easier:

- Performing a linear scan to search is feasible if the response time and the cost on expected workloads are acceptable.
- Distributing trust across multiple trust domains is acceptable if certain security requirements are met.

These opportunities serve as the basis for our system design.

2.4 Building a distributed trust system

We now discuss how to build a system where an attacker who compromises part of the infrastructure cannot easily gain access to the entire infrastructure. Such a model has already been deployed in several real systems, including cryptocurrencies relying on consensus such as Ripple [90] or Stellar [86], Certificate Transparency [81], and academic work [31].

Split across clouds. By treating different clouds as distinct trust domains, a malicious cloud provider (or an attacker that can exploit a vulnerability in one cloud infrastructure), cannot gain access to both trust domains.

Split across institutions. By using trust domains in competing organizations or nonprofits generally trusted by the public (e.g., the Electronic Frontier Foundation), users can have a stronger assurance that the organizations are unlikely to collude.

Split across jurisdictions. By separating trust domains by jurisdiction (i.e. different countries), a single legal authority cannot gain access to the entire system.

If the trust domains are deployed in the cloud, we can take advantage of the fact that cloud providers are monetarily incentivized to provide availability. Fail stops can still occur naturally, but cloud providers make it easy to detect failures and launch new servers. Clients can report statistics on the lack of availability of a trust domain, and the organization deploying the system can take its business elsewhere.

2.5 Future directions

As we conducted our survey, some companies mentioned additional features that, while not necessary for initial deployment, are desirable. Although we do not support these in DORY, we note them here as potential directions for future work.

Concentrate resources in a single trust domain. The trust domain already used for the filesystem should do most of the work for search as well. Each additional trust domain should do little work, so that adding a new trust domain should be cheap. DORY concentrates resources to some extent, (§5), but, as discussed in §4, still requires a server in each trust domain to perform a linear scan.

Richer search functionality. Several companies mentioned that they would appreciate richer search functionality beyond

simple keyword search (e.g. ranked search based on term frequency.) DORY only returns the set of documents containing a keyword, leaving ranked search for future work.

3 System design overview

In DORY, we focus only on the search system for end-to-end encrypted filesharing systems and not on the design of these filesharing systems. These systems [73, 107, 115, 121, 124] already exist and are in use. We design DORY to build on top of and interface with these systems as described in §3.2. For this purpose, we abstract out the underlying filesystem.

3.1 The underlying filesystem

End-to-end encrypted filesystems (including the five companies we surveyed in §2) tend to follow a common design pattern, which we now describe. To hide the contents (including the name) of documents, these filesystems assign a document ID to each document and associate the ID with an encryption of the document contents. Documents accessible by the same users are grouped into *folders*, each of which has a corresponding ID. Users who have access to the same folder share a (logical) secret key used to encrypt the documents in that folder. In this way, while the server learns the IDs of documents being accessed, the number of documents in each folder, and which users have access to which folders, it does not see the contents of the documents.

When a user is added to a folder, the other users share the existing folder key with the new user, and when a user's access to a folder is revoked, the remaining clients choose a new folder key. To prevent the remaining clients from having to re-encrypt every document in the folder after a user is revoked, these systems employ lazy revocation (as described in §2.1).

Users may choose to keep some documents synchronized with the server (i.e., store the most recent version of the document locally) and others not synchronized (i.e., do not store locally and retrieve them from the server only as needed). In either case, the user has already downloaded the most recent version of the document before she sends an update. In the case where two clients try to update the same file simultaneously, these systems often create two versions of a file.

DORY integrates with the filesystem (FS) using the following FS API (depicted in Figure 3):

- getCurrKey(folderID) $\rightarrow k$: Get the current key associated with the group of files in folderID.
- getDocKey(docID) $\rightarrow k$: Get the key used in the most recent update for docID.
- getDocIDs(folderID) $\rightarrow$ docIDs: Get all the document IDs used for the documents in folderID.
- getVersion(folderID, docID) $\rightarrow$ version: Get the current version number associated with a file.

3.2 The DORY API

When a user searches or updates a file, the filesystem client calls the DORY client via DORY's API so that DORY performs the search or incorporates new updates into the search index. We now describe DORY's client API, depicted in Figure 3.

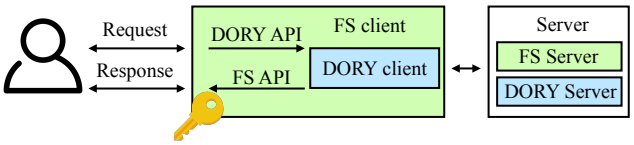


Figure 3: System software architecture. The figure shows the structure of the software rather than the physical system itself, where the server is instantiated across multiple machines.

When the user updates a document in the underlying filesystem, the user's client also sends an update to the DORY client to maintain the search index, allowing DORY servers to respond to subsequent search queries correctly.

The underlying filesystem already handles key management by giving permitted users access to the folder key(s). DORY leverages this key management mechanism so the permissions of the filesystem naturally extend to DORY: when a user is added to or removed from a folder in the underlying filesystem, she also gains or loses the ability to search in DORY.

We also utilize the fact that to update a document in the underlying filesystem, the user has already downloaded that document (if it is not being added for the first time). We employ the conflict-resolution mechanisms in the underlying filesystem to resolve conflicts in search index updates.

DORY exposes the following API to filesystem clients:

- Update(folderID, docID, prevWords, currWords): Given the folder ID, the document ID of a document in that folder, the previous set of keywords in that document prevWords, and the current set of keywords in that document currWords, update the state at the DORY servers.
- Search(folderID, keyword) $\rightarrow$ docIDs: Given the folder ID to search over and a keyword, find all the documents containing that keyword. DORY has a small (configurable) false positive rate, but DORY has no false negatives.

Updates require the client to upload a small, constant-sized amount of data per file, and searches require the server to perform a linear scan over the search index for a given folder (the cost of search for a user only depends on the number of files that user has access to).

3.3 System architecture

Folders in DORY are divided into *partitions*, each of which is managed by a different group of servers. A deployed system may contain many such partitions, and execution across partitions occurs in parallel. The following entities comprise DORY's system architecture for a single partition (Figure 4):

- **Filesystem server:** The underlying filesystem provides the functionality described in §3.1.
- **Replicas:** The ℓ DORY replicas maintain identical copies of the search index and execute search queries. Each replica is deployed in a separate trust domain. In our implementation, we use $\ell = 2$.

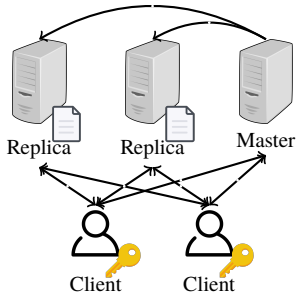


Figure 4: DORY's physical system architecture for a single partition (filesystem server not pictured). Replicas should be deployed in different trust domains, and each holds a copy of the search index.

- **Master:** The DORY master ensures that the ℓ replicas have the same view of the state and that the clients know the version of this state and which servers to contact. The master can be deployed in any existing trust domain.
- **Clients:** Multiple clients send requests to the filesystem server and the DORY master and replicas. Each client only needs to store three 128-bit keys (and can optionally cache version numbers received from the master).

To search, the client must interact with ℓ replicas for each partition. The master can be co-located with the filesystem server to ensure that updates to the search system and underlying filesystem occur atomically, although this is not necessary.

3.4 Threat model and security properties

We now describe DORY's security properties at a high level, and include DORY's formalism (detailing the guarantees) and proof in the full version [36]. In short, we achieve the security goals in §2.2. We discuss security at the level of trust domains, each of which may deploy one or more servers.

Below, we assume that the underlying filesystem is maliciously secure. In particular, we assume that DORY's client can always retrieve the correct version number from the underlying filesystem. Providing such a guarantee (e.g., by detecting rollback and fork attacks in filesystems) is a well-studied line of work [11, 63, 70, 75, 82]. If the underlying filesystem only defends against an honest-but-curious attacker, though, DORY also only protects against such an attacker.

Security with one honest trust domain. A malicious attacker that compromises $\ell - 1$ of the ℓ trust domains does not learn any search access patterns. More precisely, such an attacker learns nothing except what is leaked by the underlying filesystem, as well as the timing of individual search requests and the folders they take place over. This security property implies both forward privacy, the privacy of newly added files in the presence of previous queries, and backward privacy, the privacy of deleted files after deletion, as defined by Stefanov et al. [116]. Notably, we do not leak the number of search results; if leaked, this information could open the door to volume-based attacks [102] (parameters that determine result sizes are public).

Security with no honest trust domains. DORY's goal is to hide search access patterns when at least one trust domain is honest. When all trust domains are compromised, we have the modest goal of defaulting to the security of prior schemes leaking search access patterns, instead of readily losing all security by immediately exposing the search index. In this case, the only additional leakage (on top of what the attacker learns if at least one trust domain is honest) is a deterministic identifier for the keyword queried. In the security definition for our cryptographic protocol, we model the attacker as seeing queries only after the point of compromise; in reality, systems retain leakage (e.g. cache state) that increases the amount of information the attacker can access [57].

We formally model the end-to-end security guarantees of DORY for the case where at least one trust domain is honest and the case where no trust domains are honest by defining an ideal functionality $\mathcal{F}$ that specifies the behavior of an ideal system, capturing the properties discussed above. $\mathcal{F}$ further captures the fact that the client can verify the integrity of the result. In the full version [36], we present a formal definition using $\mathcal{F}$ and prove the following theorem, which captures DORY's security:

Theorem 1: *Using the definitions in the full version [36], DORY securely evaluates (with abort) the ideal functionality $\mathcal{F}$ when instantiated with a secure PRF, a secure aggregate MAC, a secure distributed point function, and a secure filesystem that implements the ideal filesystem functionality.*

DORY does not provide availability if any one trust domain refuses to provide service (see §2.4 for how cloud providers are monetarily incentivized to provide availability).

Relationship with underlying filesystem. DORY interfaces with deployed end-to-end encrypted filesystems (§3.1). These, as mentioned, allow the server to learn the ID of the file being accessed (but not its contents). While search itself is protected in DORY, some side effects of the search results are not: If, after seeing the search results, a user decides to open (and retrieve from the filesystem) a file in the results, an attacker could infer that the file matched the search. DORY does not address these side effects, but simply aims to not add any leakage to the overall system during search. These side effects (and leakage due to the filesystem) can be prevented by running DORY on top of an oblivious filesystem.

Extension to oblivious filesystems. Some file storage proposals [10, 26, 58, 91, 92] hide which files are being accessed. These are usually based on oblivious algorithms [119], which have significant overhead and have not yet been deployed. Nevertheless, in §4.5, we discuss how DORY can be used to provide search for an example of such a filesystem design, demonstrating that DORY's techniques do not require the server to know the file ID being updated.

4 Search design

We start by describing a basic encrypted search scheme that leaks search access patterns and is only secure against an honest-but-curious attacker in §4.1. We will show how to modify our basic scheme to eliminate search access patterns

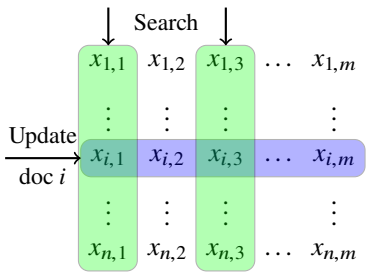


Figure 5: Search index layout for n documents with Bloom filters of length m. Updates write *rows* and searches retrieve *columns*.

in §4.2, move from an honest-but-curious to malicious threat model in §4.3, and support dynamic membership in §4.4. We show the pseudocode for the complete search protocol in the full version [36]. For simplicity, we only discuss search servers, which we assume are deployed in different trust domains, and ignore the master and filesystem servers in this section.

4.1 A strawman search index

In our initial version, clients have access to a single server. For every document, the server stores an encrypted Bloom filter corresponding to the set of keywords in the document. To update the search index for a particular document, the client computes the Bloom filter for the contents of the document and encrypts it using a one time pad unique to that update. We generate the mask for a document using a pseudorandom function (PRF) keyed with a per-folder key and the current document version number as input. The key management functionality built into the underlying filesystem ensures that every client has a copy of this PRF key.

If there are n documents in the search index and Bloom filters are m bits, then we can think of the server as storing an $n \times m$ table where each element is a single bit (Figure 5). Each row in the table is a Bloom filter for a document, and the ith row corresponds to the document with ID i. For an update, the client sends a new *row* that the server inserts into its table. This allows the client to easily modify existing documents and add new ones: the server either replaces an existing row with the new row or appends the new row to the table.

To search for a keyword, the client must find all the documents where the Bloom filter indexes corresponding to that keyword are set to "1". The client can check this by retrieving from the server the *columns* corresponding to the Bloom filter indexes for that keyword. The client can decrypt bit b_i in a column by computing the mask for row i, extracting the mask bit corresponding to that column r_i, and then evaluating $b_i \oplus r_i$. If the ith entry in each of the decrypted columns is set to "1", then the client marks document i as containing the keyword. In order to prevent the attacker from learning the queried keyword from the requested indexes, we compute the Bloom filter indexes using a PRF keyed with a per-folder key and the keyword as input. This key is managed by the underlying filesystem in the same way that the other PRF key is.

We note that in order for the contents of the client's update to remain hidden from the server, the client must be able to retrieve the correct version number from the underlying filesystem. Without this guarantee, the client could use the same mask twice, leaking information about the update contents. For this reason, we only provide security against a malicious attacker if the underlying filesystem also provides the correct version numbers (discussed in §3.4). This strawman proposal is similar to the one described in [76].

4.2 Eliminating search access patterns

To eliminate search access patterns, we need to hide from the server which columns the client is retrieving during a search. To do this, we use a private information retrieval (PIR) protocol [27, 28], which allows a client to retrieve an entry in a database from a server (1) without the server learning which entry is being retrieved, and (2) using total communication sublinear in the database size.

Tool: Distributed Point Functions (DPFs). One efficient way to implement PIR is using a distributed point function (DPF) [51] (later generalized as function secret sharing [20, 21]), which we identify as particularly well-suited for our setting. DPFs allow a client to split a point function f into *function shares* such that any strict subset of the shares reveal nothing about f, but when the evaluations at a given point x are combined, the result is $f(x)$.

A DPF is defined by the following algorithms:

- $\mathsf{DPF.Gen}(a, b) \rightarrow (K_1, \ldots, K_\ell)$: Generates keys $K_1, \ldots, K_\ell$ that allow the ℓ servers to jointly evaluate the point function that evaluates to b at input a.
- $\mathsf{DPF.Eval}(K_i, x) \rightarrow y$: Evaluates the function share corresponding to key K_i at server i on input x to produce output y.

To evaluate the point function f where $f(a) = b$ on some input x, the client generates keys for all ℓ servers by running $\mathsf{DPF.Gen}(a, b)$ and sending K_i and x to server i for all ℓ servers. Server i then runs $\mathsf{DPF.Eval}(K_i, x)$ and returns the result y_i to the client. The client can then compute $y_1 \oplus y_2 \cdots \oplus y_\ell$ to reconstruct $f(x) = y$. We make black-box use of the construction from Boyle et al. where $\ell = 2$ [21].

Leveraging DPFs to search. To hide search access patterns, we switch from having the client interact with a single server to having the client interact with ℓ servers in different trust domains that hold identical copies of the search index. To retrieve column j, the client generates shares of the point function that evaluate to all 1's at column j and all 0's for all other columns. The client then sends a function share to each server. Each server evaluates its function share for each column, ANDing the DPF evaluation with the contents of the column, and sends the XOR of the results back to the client. The client then assembles the responses to recover column j.

Using DPFs to retrieve columns requires a linear scan over the search index for a folder. While this is expensive asymptotically, we only aim to show efficiency for realistic workloads, motivating our decision to compress the search index using Bloom filters.

4.3 Protecting against malicious attackers

So far, we have assumed that all servers are honest-but-curious. We now show how to defend against a malicious attacker (namely, an attacker that can deviate from the protocol) that can compromise up to $\ell - 1$ of the ℓ servers. To achieve this, we need to ensure that for a search, the server evaluates the DPF on columns corresponding to the most recent updates sent by the client (not corrupted or old updates).

Strawman: MAC for every bit. We start by showing a strawman that employs MACs, but increases the bandwidth and search latency by roughly a factor of the MAC tag size (typically 256). For each update, the client additionally sends a MAC tag for every bit in the encrypted Bloom filter. The client cannot send a single tag for the row because to search, the client must retrieve individual columns rather than entire rows. We can think of the server as now storing a second table of MAC tags where each entry of this table is the tag for the corresponding entry in the original table (as in Figure 5).

We need to ensure that (1) a tag is only valid for a particular document update (to prevent replay attacks) and that (2) it cannot correspond to a different Bloom filter index. To do this, we compute the MAC over not only the single Bloom filter bit, but also the document ID, Bloom filter index, and document version number. As with the PRF key, we use the key management functionality in the underlying filesystem to ensure that every client has a copy of the MAC key.

The client now runs the DPF over the columns in both the original table and the MAC tag table. After assembling the responses from all ℓ servers, the client can check that the tag for every bit is correct. However, this increases both the bandwidth and the time to perform the linear scan over the index (i.e., the search latency) by a factor of the tag size. We identify aggregate MACs as a tool to transform this factor from a multiplicative to an additive one.

Tool: Aggregate MACs. We leverage aggregate MACs [71] to allow the servers to combine individual MAC tags into a single aggregate MAC tag. Aggregate MACs, analogous to aggregate signatures [17], allow multiple MAC tags computed with possibly different keys on multiple, possibly different messages to be aggregated into a shorter tag that can still be verified using all the keys. Notably, aggregating MAC tags does not require access to the keys.

The Katz-Lindell aggregate MAC construction [71] works as follows. To generate a MAC tag for some message m using a key k, we simply use a pseudorandom function MAC and compute $t \leftarrow \mathsf{MAC}(k, m)$. To aggregate MAC tags $t_1, \ldots, t_n$, the aggregator computes $T \leftarrow \oplus_{i=1}^{n} t_i$. To verify an aggregate MAC tag T using messages $m_1, \ldots, m_n$ and keys $k_1, \ldots, k_n$, the verifier checks $T \stackrel{?}{=} \oplus_{i=1}^{n} \mathsf{MAC}(k_i, m_i)$.

Aggregating MAC tags to improve performance. To improve performance by a factor of the tag size, we allow the servers to combine individual tags into a single aggregate tag. To search, the server evaluates the DPF on the contents of the column and a single aggregate tag for the entire column.

Aggregating MAC tags also allows us to reduces storage space at the servers. Rather than storing an entire separate MAC table, the servers instead keep an array of aggregate tags, one for each column. On each update, the client XORs the old tag with the new tag (which is why Update takes both prevWords and currWords). By then XORing this value with the aggregate tag, the server can remove the old tag and add the new tag. To ensure that this aggregate MAC tag is maintained correctly, the server must check that the client has the latest version of the document; otherwise it rejects the update.

4.4 Supporting dynamic membership

Users might be added to or removed from a folder, requiring the new group to generate a new key. This new key might be in use at the same time that some parts of the search index were generated using an old key in order to support lazy revocation. We let the underlying filesystem handle key management, but we need to ensure that our search protocol supports multiple keys that may be active at the same time.

Decrypting search results is straightforward; to decrypt the results for an individual document, the client uses the same key from the last update to that document. Aggregating MAC tags is also simple because we can aggregate tags computed with different keys. We can remove old tags and add new tags with different keys using XOR in the same way as before.

4.5 Generalizing to oblivious filesystems

We briefly discuss how DORY is compatible with a filesystem that hides which document is being accessed within a folder, showing that DORY does not inherently require knowledge of which document is being accessed.

We can build a filesystem that hides document access patterns using PathORAM [119], which acts as an oblivious key-value store for each folder. To support multiple users, we keep an encrypted copy of the ORAM client state at the server (discussed in §7.1). Each ORAM block contains the encrypted contents of a document.

One straightforward way to search over this filesystem would be store an inverted index in ORAM. This would hide which document is being updated, but updates would require an ORAM access for every word in the document.

Instead, we apply DORY to this filesystem. Rather than storing encrypted Bloom filters in a table as in §4.1, we store them in a second PathORAM to hide which document is being updated. We use the same techniques for supporting multiple users as in the underlying filesystem.

To perform an update, the client generates an encrypted Bloom filter as before and needs to insert it into the ORAM index. This creates a new challenge, because ORAM accesses require the client to re-encrypt other ORAM blocks, and standard symmetric key encryption breaks DORY's column alignment. To address this, we keep track of a new value shared among users for each document: the ORAM access number, which is incremented after each ORAM access. Instead of

generating PRF masks using the document's version number, we now generate them using the document's ORAM access number, allowing clients to safely re-encrypt Bloom filters.

To execute a search, the client still generates a DPF query for the Bloom filter indexes in question and the server still needs to perform a linear scan over the search index (we must scan over every bit in every Bloom filter). Another challenge arises, because while the order of the scan was obvious when the search index was a table, the order is less obvious for the tree structure of PathORAM. We solve this problem by traversing the tree in a fixed order to generate a table layout. The client can interpret the results by reconstructing the traversal order using the position map stored as part of the ORAM client.

5 Replication across trust domains

DORY requires that the servers processing search requests operate on the same version of the index in order for the client to receive a valid response; otherwise, the cryptographic shares from the DPF cannot be combined correctly. Because our system processes a mix of update and search requests, the servers need to agree on the index state. The client also needs to know the document version numbers corresponding to the index that the servers used to execute the search; otherwise, the client will be unable to decrypt and verify the result.

Because we are in an adversarial environment, a natural solution is to use a Byzantine fault-tolerant (BFT) consensus algorithm [1, 16, 24, 33, 77, 79] to agree on the ordering of update and search requests. Standard BFT provides the properties we need, but requires $3f+1$ servers, each in its own trust domain, to handle f failures. A large number of trust domains is expensive to maintain and difficult to deploy, increasing the overall system cost. We make several observations about our setting that allow us to use only $f+1$ trust domains.

Observations we leverage. We make three observations that allow us to tailor the problem of consensus to DORY:

DORY deterministically detects server misbehavior. Our cryptographic protocol already defends against malicious servers; if a server executes the client's query incorrectly or over an incorrect version of the index, the client will detect this (triggering a manual investigation). This is a significant departure from the Byzantine fault model where failure information is imperfect. By handling server misbehavior at the cryptographic protocol layer, we can use a fail-stop rather than Byzantine failure model at the consensus layer. This and the next observations allow us to use just $f+1$ trust domains to tolerate f failures.

Trust domains provide availability. To support search, DORY needs all $f+1$ replicas to be available. We need to ensure that servers across multiple trust domains remain online to allow clients to search. Here we leverage the observation that for trust domains deployed in the cloud, the cloud provider is monetarily incentivized to provide availability (§2.4). This means that if a server in a trust domain fails, either it will eventually come back online or another server will take its place; even if failures occur, $f+1$ servers will be available again at some point in the future.

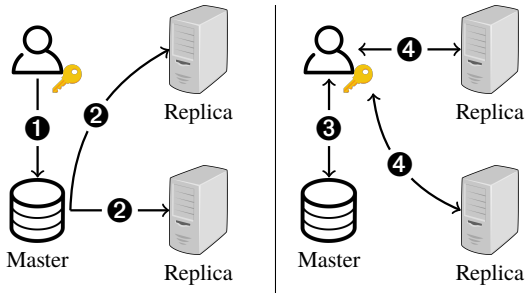


Figure 6: System architecture and protocol flow for updates (left) and searches (right). ❶ Client sends update to master. ❷ Master propagates updates to replicas. ❸ Client requests version number(s) from master. ❹ Client splits search request across replicas.

DPFs give us replication for free. The challenge now is to reinitialize the state of these failed servers. The use of DPFs in our cryptographic protocol requires all replicas to have identical copies of the search index. Normally it is unsafe to transfer state between trust domains, as the recipient has no way to verify correctness. However, because the client can check the integrity of the state used to execute a search query, we can safely copy state across trust domains. Because we have $f+1$ servers, at least one server will always remain online to preserve the state of the index.

5.1 Algorithm

A DORY cluster contains the following entities (Figure 6):

Master: The master receives updates and manages replica state. The master stores the most recent updates and version numbers (both the overall system version number and individual document version numbers), but not the entire search index. The master can be deployed in any trust domain, as clients can detect misbehavior when verifying search results.

Replicas: The replicas receive updates from the master and perform searches from the user. The replicas store the most recent versions of the index as well as the version numbers (both the overall system version number and individual document version numbers). We must deploy ℓ replicas in ℓ different trust domains to ensure that the client can split its search request across different trust domains. However, the total number of replicas n may be greater than ℓ in order to improve fault-tolerance.

We additionally use a watchdog service (commonly available in the cloud) that periodically checks that all servers are still online and triggers recovery when it detects a crash.

Properties. Our replication algorithm should provide the following properties:

- **Correctness:** If *all* of the replicas and the master fail, a client with the correct set of document version numbers can detect this.
- **Fault-tolerance:** If at most $n-1$ of the n replicas fail, then the search index is preserved. If the master fails, then the

most recent set of updates can be recovered with help from the client.

We do not guarantee availability if individual trust domains do not provide availability.

Algorithm. We now explain how we handle updates and searches and recover from failure (see Figure 6).

Updating a document. To update a document, the client sends the update along with the new document version number to the master. The master needs to send the update to the replicas and increment the version number. Because the master might fail while sending the update to the replicas, the master runs two-phase commit [80] with the replicas to ensure that all the replicas receive the update and associated version number. We do not need to worry about replica failures during two-phase commit (and so do not need multiple replicas in each trust domain); if a replica fails, the watchdog service will detect this and coordinate recovery as described below.

Searching for a keyword. To search for a keyword, the client first needs to learn the current version numbers from the master (both the overall system version number and the corresponding individual document version numbers). If the client has a relatively recent set of document version numbers, the master can simply send updates for a few of the document version numbers, making the overall bandwidth much smaller than the number of documents. The client then generates a search query for ℓ of the replicas. The replicas execute the search on the version of the index corresponding to the system version number sent by the client.

Coordinating recovery. We rely on the watchdog service to detect failures. If at least ℓ of the replicas across ℓ different trust domains remain online, clients can continue searching. Otherwise, we can start new replicas and transfer the state from a remaining replica to the new replica, even if the replicas are in different trust domains. This will cause a slight delay for clients waiting to search, but is safe due to the underlying cryptographic protocol (as discussed above). We do not need to worry if the master fails, because the master does not respond to the client until it has propagated the update to the replicas. If a replica fails during two-phase commit, the master can roll back the two-phase commit and then start another replica in the same trust domain and copy the state across trust domains.

5.2 Batching

Rather than running two-phase commit between the master and replicas for every update, we can apply batching to amortize the cost. Instead of immediately sending an update to the replicas, the master aggregates a batch of updates and, when this batch reaches a certain size or a certain amount of time has elapsed, it runs two-phase commit with the replicas to transfer the current batch of data.

However, now that the master is responding to clients before sending the updates to the replicas, we need to ensure that the master does not lose state when it fails. In particular, the master needs to be able to recover the updates that were waiting to be committed to the replicas. The master does this by comparing the individual document version numbers at the replicas with those at the filesystem server. For each document where the version numbers differ, the master can request an update from the next client to come online with access to that document.

6 Implementation

We implemented DORY in ~5,000 lines of C (for the distributed point function and other low-level cryptographic operations) and Go (for the networking and consensus). We used the OpenSSL library, and our DPF implementation closely follows the one in Express [43]. We instantiate the PRF using AES. We also implemented the DORY client on an Android Google Pixel 4. In addition to the C code, which we ported to the mobile platform, we wrote ~1,200 lines of Java. We used the tiny AES library [123] to minimize memory usage in our mobile implementation. Our implementation supports a single folder and does not include the watchdog service and coordinated recovery described as part of §5 or the generalization to oblivious filesystems described in §4.5. The source code is available at `https://github.com/ucbrise/dory` (see Appendix A for details).

6.1 Parallelism

The linear scan over the search index can be easily parallelized across both cores and servers because it carries no state from document to document.

Thread-level parallelism. Since we evaluate the DPF on each column of the search index, we parallelize the scan operation by simply assigning each thread a number of columns and then combining the results computed by each thread.

Server-level parallelism. We can partition the search index by having different pairs of replicas maintain different parts of the search index. The client then sends a search query to all pairs of replicas and simply computes the union of the results. Replica partitioning improves latency since each replica now only needs to search over a part of the index instead of the full index. Each pair of replicas can store part of the search index for many folders, making it possible to keep search latency low, but the overall throughput high.

6.2 Fast PRF evaluation

In order to decrypt the search result received from the server, the client must compute a mask for each individual document. To reduce the number of PRF evaluations to decrypt, we group Bloom filter indexes for the same keyword in the same 128-bit block. This grouping allows the client to decrypt the search results for one document using a single PRF evaluation. This does not significantly impact the false positive rate of the Bloom filter because we can now model a m-bit Bloom filter storing w words as $m/128$ independent Bloom filters each storing $128w/m$ words.

7 Evaluation

We evaluated DORY to determine (1) how it performs in comparison to existing techniques and (2) whether it meets

Table 7: On the left, Bloom filter sizes (in bytes) necessary for > 1 expected false positive assuming an average of 73.18 keywords per document where each keyword hashes to 7 Bloom filter indexes (Table 7a). On the right, breakdown of search latency without parallelism and end-to-end search latency with parallelism where p is the degree of server parallelism (Table 7b).

Docs	BF size
$\leq 2^{10}$	140 B
$\leq 2^{11}$	160 B
$\leq 2^{12}$	180 B
$\leq 2^{13}$	200 B
$\leq 2^{14}$	225 B
$\leq 2^{15}$	250 B
$\leq 2^{16}$	280 B
$\leq 2^{17}$	315 B
$\leq 2^{18}$	350 B
$\leq 2^{19}$	390 B
$\leq 2^{20}$	435 B

(a)

Docs	Time breakdown, p=1 (ms)				End-to-end latency (ms)		
	Consensus	*Client*	*Network*	*Server*	*p=1*	*p=2*	*p=4*
2^{10}	0.73	0.54	58.67	2.68	62.62	61.81	61.51
2^{11}	0.73	0.87	58.41	4.11	64.12	62.39	61.89
2^{12}	0.73	1.52	57.99	7.09	67.33	64.46	62.92
2^{13}	0.73	2.80	58.74	12.03	74.30	68.08	64.78
2^{14}	0.75	5.30	77.88	26.24	110.17	75.76	68.59
2^{15}	0.76	10.18	80.59	50.97	142.50	112.71	76.76
2^{16}	0.81	19.83	100.67	108.78	230.09	147.39	115.50
2^{17}	0.86	38.99	119.38	240.45	399.48	243.43	153.56
2^{18}	1.19	76.92	142.28	527.67	748.06	428.40	256.15
2^{19}	1.78	154.37	151.98	1172.46	1480.59	800.98	454.52
2^{20}	2.81	306.34	148.96	2602.83	3060.94	1636.80	862.42

(b)

the requirements outlined by the companies we surveyed. We consider the following metrics: latency (§7.2), throughput (§7.3), storage (§7.4), bandwidth (§7.5), and cost (§7.6). We compare DORY's performance to two different variations of DORY as well as plaintext search and a baseline built on ORAM (§7.1) that provides similar guarantees to those of DORY. We show that DORY meets the requirements outlined by the companies we surveyed and outperforms (in some cases, by orders of magnitude) our ORAM baseline (§7.1).

Experimental setup. We evaluate DORY on AWS using r5n.4xlarge instances with 128GB of memory and 16 CPUs for the replicas and the master. We use a c5.large client with 4GB of memory and 2 CPUs to model a user's desktop machine. We use an Android Pixel 4 to measure the time to search on a mobile client. We place the two trust domains in different regions (`east-1` and `east-2`) to ensure that machines are in different clusters to model different organizations, although in practice these clusters would likely be geographically close to maximize performance. All communication occurs over TLS. We run experiments for a single folder; a real system would maintain many such folders in parallel.

System parameters from Enron email dataset. We use the Enron email dataset, which is commonly used to evaluate searchable encryption schemes [22, 65, 84, 94, 96, 97, 129] to set Bloom filter sizes for DORY. We leverage the same standard keyword extraction techniques used in Oblix [94]: we stemmed the words and removed stopwords and words that were > 20 or < 4 characters long or contained non-alphabetic characters. In the over 500K emails, each email has an average of 73.18 keywords with a standard deviation of 114.89.

Regarding the configuration of the Bloom filters, each keyword hashes to 7 locations in the Bloom filter, as we found that it provided a reasonable tradeoff between the time to perform the linear scan at the server and bandwidth. We choose the Bloom filter size based on the number of documents in a folder so that, for every search in that folder, the search results have less than one false positive document in expectation. The sizes of the Bloom filters are specified in Table 7a.

7.1 Baselines

We evaluate DORY in comparison to four baselines:

- **ORAM baseline**: Eliminates search access patterns using ORAM (expected to incur a significant overhead). With this baseline, we show how DORY compares to a solution that provides comparable security guarantees.
- **Plaintext search**: Searches over a plaintext inverted index and does not provide any security guarantees (expected to have much lower overhead than DORY).
- **Semihonest DORY**: Modifies the DORY protocol to only provide security against semihonest adversaries (expected to have lower overhead than DORY).
- **Leaky DORY**: Modifies the DORY protocol to allow search access pattern leakage by using only one trust domain and querying the replica directly for the indexes corresponding to a keyword rather than using a DPF (expected to have lower overhead than DORY).

Semihonest DORY illustrates the overhead of the MAC checks necessary to defend against malicious adversaries, and leaky DORY illustrates the overhead of the DPF queries. In all of the baselines except the ORAM baseline, we use the same consensus system as in DORY, although for the baselines where there is only one trust domain (leaky DORY and plaintext search), the master only needs to send update batches to a single trust domain (we model this by placing all servers in the same AWS region). Only the ORAM baseline has security guarantees comparable to those of DORY.

ORAM baseline. Many academic works [61, 65, 96, 116] point to an inverted index in ORAM [54, 99] as a way to achieve searchable encryption without search access pattern leakage, making it a natural baseline for searching within a folder. Traditional ORAM is designed for a single client and requires the client to maintain ORAM client state hidden from the server [119]. A separate line of work explores extending single-user constructions to multi-user settings [10, 26, 58, 91–93]. Mayberry et al.'s system [93] is particularly fit for our setting as it protects mutually trusting clients (clients with access to a given folder) from a malicious server. For a semi-honest server or for a malicious server for which we have a mechanism to

verify the data returned (discussed in §3.4), their protocol uses a single-user ORAM and requires clients to store the encrypted ORAM client at the server. To perform an operation, the client acquires a lock at the server, downloads and decrypts the ORAM client state, performs the operation, encrypts and sends back the state, releasing the lock.

Client failures. We observed that the above proposal did not consider client failures. If a client fails after issuing operations at the server but before uploading the updated client ORAM state, the next client's access may leak search access patterns (e.g. if it searched for the same word as the previous client). To handle client failures, we require each client to record a client "prepare" operation at the server, and if it fails before completing, the next client can finish the operation.

Eliminating frequency leakage. Popular keywords require multiple ORAM blocks to store all the document identifiers containing that keyword. We need to ensure that the number of blocks accessed doesn't leak the frequency of a keyword due to known attacks [102], as DORY does not leak this frequency. For each search, we fetch the maximum number of blocks a keyword maps to. Similarly for each keyword we update in a document, we fetch the maximum number of blocks a keyword maps to and write back a single block.

Implementation. We implemented our baseline on top of an existing open-source PathORAM implementation in Go [101].

Evaluation on Enron email dataset. While DORY's performance relies only on the system parameters and not the contents of the documents themselves, the performance of both our ORAM and plaintext search baselines depends on document contents. We evaluate these baselines using subsets of the Enron email dataset with the same keyword extraction techniques described above. To evaluate different numbers of documents, we take different-sized subsets of the Enron email dataset. We treat updates as adding an entire email to the index. Because the Enron email dataset only has ~ 528K emails, we do not measure the ORAM and plaintext search baseline beyond that number of documents.

7.2 Latency

Update latency. Figure 8 shows that the update latency of DORY is orders of magnitude faster than that of the ORAM baseline. This holds for both the desktop and mobile clients (Figure 9). The baseline requires a number of ORAM accesses (each of which necessitates a round trip) linear in the number of document keywords. In contrast, DORY simply uploads a single encrypted Bloom filter. Update latency determines (1) how long it takes for updates to be reflected in search results and (2) how long the client must remain online. Neither is a concern in DORY where updates are processed in less than 1ms, but the ORAM baseline requires clients to remain online for hours. Note that semihonest DORY has a faster update time than DORY because the client does not have to generate a MAC for every bit in the Bloom filter.

Search latency. Table 7b shows the breakdown in search

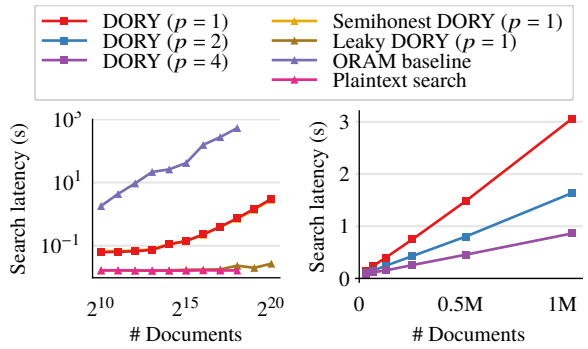


Figure 8: Search latency and update latency. The two figures on the left use a logarithmic scale on both axes, and the figure on the top right uses a linear scale on both axes (p denotes server parallelism). The update latency of leaky DORY exactly matches that of DORY, and the search latency of semihonest DORY is slightly less than that of DORY.

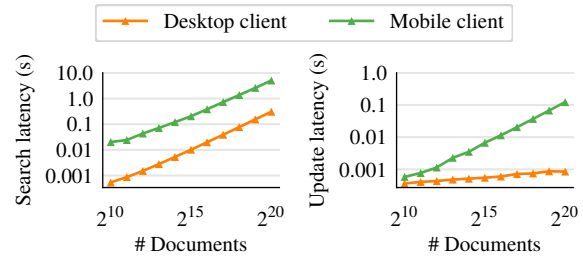


Figure 9: Latency for mobile client and desktop client. Both plots use a logarithmic scale on both axes.

latency. As the number of documents increases, the majority of time is spent performing the linear scan at the server. This is apparent in Figure 8, where leaky DORY's search latency is significantly lower than that of DORY and stays relatively constant as the number of documents increases due to the fact that leaky DORY does not need to perform a linear scan.

Despite overheads incurred due to the linear scan, DORY is orders of magnitude faster than the ORAM baseline. The MAC overhead to protect against malicious adversaries is barely noticeable, as semihonest DORY and DORY have almost identical search latencies. Mobile clients incur additional overhead in comparison to desktop clients (the mobile client spends 5 seconds on client-side processing for 1M documents). This overhead is below 1 second for 2^{17} documents (Figure 9).

By increasing the degree of parallelism p and partitioning the search index across replica groups, we can reduce the server time by roughly a factor of p, as this time is linear in the number of documents (Figure 8). Parallelism allows us to reach the target latency set by the companies (Table 2).

7.3 Throughput

DORY achieves significantly higher throughput than the ORAM baseline (Figure 10). Parallelism improves DORY's

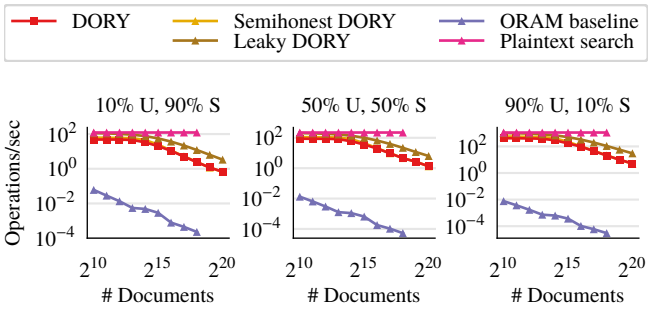


Figure 10: Throughput under a variety of workloads (U indicates updates, S indicates searches). The performance of semihonest DORY closely matches that of DORY. All plots use a logarithmic scale on both axes.

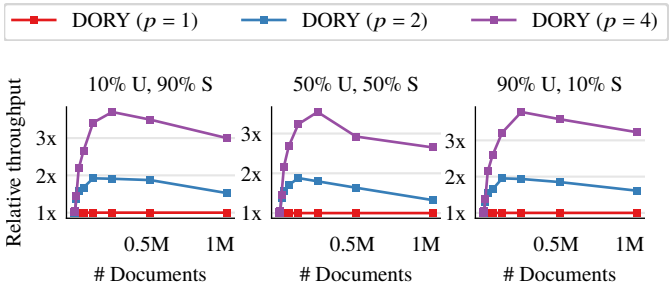


Figure 11: Effect of parallelism (p denotes the degree of parallelism) on throughput for different workloads (U indicates updates, S indicates searches).

throughput by roughly a factor of p for larger numbers of documents (Figure 11). Relative to other workloads, DORY performs best under update-heavy workloads (updates require an insertion while searches require a linear scan), and the ORAM baseline performs best under search-heavy workloads (searches require fewer ORAM accesses than updates).

7.4 Storage

Server state. Figure 12 shows how DORY uses substantially less storage space at the server than the ORAM baseline and storage space comparable to that of a plaintext inverted index. DORY's index continues to grow at a constant rate for large numbers of documents while the index for plaintext search grows more slowly, making the plaintext search index smaller than the DORY search index for larger numbers of documents.

Client state. DORY only requires that the client store three 128-bit keys. To generate an update or decrypt a search result, the client also needs to know the version number for each document. To minimize bandwidth, the client can optionally cache the latest version numbers so that it only needs to retrieve the version numbers that changed. For 45K documents (the highest average number of documents per user among the companies we surveyed), storing these version numbers would require 175.8KB. For 1M documents, storing these would require 3.84MB. Our ORAM baseline only requires the client to store a single 128-bit AES key to encrypt and decrypt the ORAM client, and plaintext search requires no client storage.

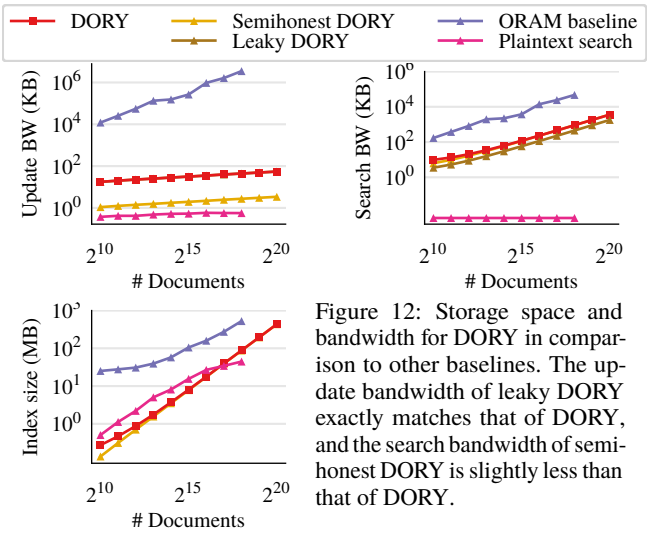


Figure 12: Storage space and bandwidth for DORY in comparison to other baselines. The update bandwidth of leaky DORY exactly matches that of DORY, and the search bandwidth of semihonest DORY is slightly less than that of DORY.

7.5 Bandwidth

Search and update bandwidth is also much smaller in DORY than in the ORAM baseline (Figure 12). The ORAM baseline incurs a significant overhead by sending the encrypted client state, but ORAM accesses are responsible for the majority of the communication. In contrast, the search bandwidth in DORY is linear in the number of documents, and the update bandwidth depends on the size of the Bloom filter. MACs are responsible for a significant part of the update bandwidth in DORY, which is why semihonest DORY has much lower update bandwidth. The difference in search bandwidth between leaky DORY and DORY is due to the size of the DPF keys; however, unlike plaintext search, the search bandwidth for both is still linear in the number of documents. We do not include the bandwidth to retrieve version numbers for individual document numbers in DORY, as these version numbers can for the most part be cached at the client as described above.

Comparison to client index. To evaluate the practicality of a client-side index instead of DORY, we built an inverted index over the Enron email dataset using a B+ tree. We found that the index is 159.9MB and while it is feasible to store this amount of data, even on a mobile device, synchronization requires significant bandwidth. One way to keep this data structure updated would be to require each client to download the contents of every update. However, this solution requires the same amount of bandwidth as syncing all the files locally, which we were trying to avoid in the first place. Instead, we could keep an encrypted copy of the client index at the server. Which part of the index is updated leaks information about the document contents, and so whenever a client performs an update, it must encrypt the entire index and send it to the server. Before a client updates or searches, it must download the most recent copy of the search index. This results in roughly a 365× increase in search bandwidth and a 3,334× increase in update bandwidth in comparison to DORY.

7.6 Cost

The companies we surveyed estimated a workload with 50% updates and 50% searches, and the highest average number of documents per user reported was 45K. The throughput of two replicas and a master operating on a folder of 45K documents under this workload is 19.5 operations/second. One of the companies reported that active users make roughly 50 updates per day, and so based on 100 operations per day and the cost to run a single r5n.4xlarge instance ($1.192/hour), each user costs roughly $0.0509 per month, well under the maximum permissible cost per user per month of $0.70-$5.54 reported by the companies. Depending on the way in which trust is distributed (see §2.4), trust domains may incur additional setup and maintenance costs not captured by our calculation.

8 Related Work

Symmetric Searchable Encryption (SSE). A long line of work has examined the problem of Symmetric Searchable Encryption (SSE) [23, 25, 35, 37–40, 50, 52, 67, 68, 97, 111, 114, 116], summarized in the following surveys [18, 59, 103]. Many of these schemes assume a single user and do not support efficient revocation, but more importantly, they permit some search access pattern leakage, opening the door to attacks [22, 65, 72, 84, 102, 106, 129]. SEAL [39] explicitly allows developers to tradeoff between leakage and performance.

Multi-server SSE and ORAM. Some SSE schemes use multiple servers to improve efficiency but still permit leakage, with some providing richer functionality than simple keyword search [15, 45, 64, 78, 100, 108]. Bösch et al. [19] and Hoang et al. [62] use multiple servers to hide search access patterns and improve efficiency. Hoang et al. [62] use a similar table layout where updates and searches correspond to different dimensions in the table. However, both schemes do not support multiple users, assume honest-but-curious servers, and require expensive updates to hide the document being updated. Our scheme also has similarities to distributed ORAM schemes that leverage multiple servers to hide access patterns with improved efficiency [3, 42, 55, 89, 117]. Implementing search with one of these schemes would still require clients to perform an ORAM access for every document keyword during an update.

Multi-user SSE and ORAM. Many existing multi-user searchable encryption schemes that support fast revocation use a different key for each user and leverage proxy encryption [8, 13] or pairings [13, 74, 104, 122]. This class of schemes use deterministic query encryption algorithms that leak search access patterns. The most efficient ORAM constructions assume a single user, with multi-user ORAMs incurring a much larger overhead by leveraging expensive tools such as multi-party computation (MPC) [10, 26, 58, 91, 92].

SSE and ORAM with trusted hardware. One way to improve performance and, in the case of search, potentially reduce leakage is by leveraging trusted hardware. ZeroTrace [113], Obliviate [5], ObliDB [44], GhostRider [83], Tiny ORAM [46], and Shroud [87] combine oblivious techniques with trusted hardware. HardIDX [49], Oblix [94], POSUP [60], and Amjad et al. [6] use trusted hardware specifically for the problem of searching on encrypted data. Unlike DORY, such solutions only require a single server, but they necessitate both additional trust assumptions (due to known side-channel attacks) and additional deployment costs.

Prior use of DPFs in systems. Splinter [125] uses function secret sharing (both DPFs and range queries) to allow users to efficiently make private queries on a public, immutable database. DURASIFT [45] uses DPFs with MPC across multiple servers to support boolean expressions of keyword searches for multiple users without leaking search access patterns. However, its techniques incur significant overhead in comparison to ours, and the authors consider thousands rather than millions of documents. Floram [42] uses DPFs to implement a distributed-trust ORAM that has linear costs but fast concrete performance. Metadata-hiding communication also benefits from DPFs (e.g. Riposte [32] and Express [43]).

BFT consensus and fault-tolerance. BFT consensus [1, 16, 24, 33, 77] is a classical problem. Prior work has explored reducing the number of participants in BFT consensus by separating agreement from execution [127], only activating some nodes when failures are detected [41, 69, 126], relaxing synchrony assumptions [2, 85, 105], adopting a hybrid fault model [105], and using an attested, append-only log [29]. A separate line of theoretical work considers Byzantine fault-tolerance specifically for the case of private information retrieval [12, 14, 47, 120] using information-theoretic tools.

Oblivious systems. ObliviStore [118], Obladi [34], Opaque [130], Cipherbase [7], and Taostore [112] are practical systems for obliviously storing and querying data (not necessarily for the problem of searchable encryption).

9 Conclusion

DORY is an encrypted search system that distributes trust to meet real-world efficiency and security requirements. By reexamining the system model, we are able to build a system that is performant without leaking search access patterns.

Acknowledgments. We would like to thank Zoë Bohn, Henry Corrigan-Gibbs, Ioannis Demertzis, Saba Eskandarian, Vivian Fang, David Mazières, Rishabh Poddar, and Wenting Zheng for providing feedback on early drafts. We also thank the leadership of Keybase, PreVeil, SpiderOak, Sync, and Tresorit for generously taking the time to meet with us and discuss their use cases. We thank the OSDI anonymous reviewers for their detailed feedback, and our shepherd Andreas Haeberlen for his working reviewing our camera-ready. This work was supported in part by the NSF CISE Expeditions Award CCF-1730628, and gifts from the Sloan Foundation, Bakar Program, Alibaba, Amazon Web Services, Ant Financial, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. This work was also supported by a NSF GRFP fellowship.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. *SOSP*, 39(5):59–74, 2005.

[2] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren. Efficient synchronous byzantine consensus. *arXiv preprint arXiv:1704.02397*, 2017.

[3] I. Abraham, C. W. Fletcher, K. Nayak, B. Pinkas, and L. Ren. Asymptotically tight bounds for composing ORAM with PIR. In *PKC*, pages 91–120. Springer, 2017.

[4] S. Ackerman. Lavabit email service abruptly shut down citing government interference, 2013. https://www.theguardian.com/technology/2013/aug/08/lavabit-email-shut-down-edward-snowden.

[5] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *NDSS*, 2018.

[6] G. Amjad, S. Kamara, and T. Moataz. Forward and backward private searchable encryption with SGX. In *Proceedings of the 12th European Workshop on Systems Security*, pages 1–6, 2019.

[7] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[8] M. R. Asghar, G. Russello, B. Crispo, and M. Ion. Supporting complex queries and access policies for multi-user encrypted databases. In *Workshop on Cloud computing security workshop*, pages 77–88. ACM, 2013.

[9] M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In *ESORICS*, pages 327–346. Springer, 2006.

[10] M. Backes, A. Herzberg, A. Kate, and I. Pryvalov. Anonymous ram. In *ESORICS*, pages 344–362. Springer, 2016.

[11] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. {SPEICHER}: Securing lsm-based key-value stores using shielded execution. In *FAST*, pages 173–190, 2019.

[12] K. Banawan and S. Ulukus. The capacity of private information retrieval from byzantine and colluding databases. *IEEE Transactions on Information Theory*, 65(2):1206–1219, 2018.

[13] F. Bao, R. H. Deng, X. Ding, and Y. Yang. Private query on encrypted data in multi-user settings. In *Information Security Practice and Experience*, pages 71–85. Springer, 2008.

[14] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. In *International Conference on Security in Communication Networks*, pages 326–341. Springer, 2002.

[15] S. M. Bellovin and W. R. Cheswick. Privacy-enhanced searches using encrypted bloom filters. *IACR Cryptology ePrint Archive*, 2007.

[16] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[17] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, pages 416–432. Springer, 2003.

[18] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):1–51, 2014.

[19] C. Bösch, A. Peter, B. Leenders, H. W. Lim, Q. Tang, H. Wang, P. Hartel, and W. Jonker. Distributed searchable symmetric encryption. In *PST*, pages 330–337. IEEE, 2014.

[20] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *EUROCRYPT*, pages 337–367. Springer, 2015.

[21] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *CCS*, pages 1292–1303. ACM, 2016.

[22] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679. ACM, 2015.

[23] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, volume 14, pages 23–26. Citeseer, 2014.

[24] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[25] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ASIACRYPT*, pages 442–455. Springer, 2005.

[26] W. Chen and R. A. Popa. Metal: A metadata-hiding file sharing system. In *NDSS*, 2020.

[27] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, pages 41–50. IEEE, 1995.

[28] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–982, 1998.

[29] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.

[30] W. Cohen. Enron email dataset, 2015. http://www.cs.cmu.edu/~enron/.

[31] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, pages 259–282, 2017.

[32] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Security & Privacy*, pages 321–338. IEEE, 2015.

[33] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, pages 177–190, 2006.

[34] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, pages 727–743, 2018.

[35] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[36] E. Dauterman, E. Feng, E. Luo, R. A. Popa, and I. Stoica. DORY: An encrypted search system with distributed trust. *IACR Cryptology ePrint Archive*, 2020:1280, 2020.

[37] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[38] I. Demertzis, D. Papadopoulos, and C. Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *CRYPTO*, 2018.

[39] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, 2020.

[40] I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD*, 2017.

[41] T. Distler, C. Cachin, and R. Kapitza. Resource-efficient byzantine fault tolerance. *IEEE transactions on computers*, 65(9):2807–2819, 2015.

[42] J. Doerner and A. Shelat. Scaling oram for secure computation. In *CCS*, pages 523–535. ACM, 2017.

[43] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. *arXiv preprint arXiv:1911.09215*, 2019.

[44] S. Eskandarian and M. Zaharia. ObliDB: oblivious query processing for secure databases. *VLDB*, 13(2):169–183, 2019.

[45] B. H. Falk, S. Lu, and R. Ostrovsky. Durasift: A robust, decentralized, encrypted database supporting private searches with complex policy controls. In *WPES*, pages 26–36, 2019.

[46] C. W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, D. Serpanos, and S. Devadas. A low-latency, low-area hardware oblivious RAM controller. In *FCCM*, pages 215–222. IEEE, 2015.

[47] R. Freij-Hollanti, O. W. Gnilke, C. Hollanti, and D. A. Karpuk. Private information retrieval from coded databases with colluding servers. *SIAM Journal on Applied Algebra and Geometry*, 1(1):647–664, 2017.

[48] K. E. Fu. *Group sharing and random access in cryptographic storage file systems*. PhD thesis, Massachusetts Institute of Technology, 1999.

[49] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi. HardIDX: Practical and secure index with SGX. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 386–408. Springer, 2017.

[50] S. Garg, P. Mohassel, and C. Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *CRYPTO*, pages 563–592. Springer, 2016.

[51] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *EUROCRYPT*, pages 640–658. Springer, 2014.

[52] E.-J. Goh et al. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[53] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, volume 3, pages 131–145, 2003.

[54] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[55] S. D. Gordon, J. Katz, and X. Wang. Simple and efficient two-server ORAM. In *ASIACRYPT*, pages 141–157. Springer, 2018.

[56] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *SRDS*, pages 189–198. IEEE, 2006.

[57] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *HotOS*, pages 162–168, 2017.

[58] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs. Private anonymous data access. In *EUROCRYPT*, pages 244–273. Springer, 2019.

[59] A. Hamlin, N. Schear, E. Shen, M. Varia, S. Yakoubov, and A. Yerukhimovich. *Cryptography for big data security*. Taylor & Francis LLC, CRC Press, 2016.

[60] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *PETS*, (1):172–191, 2019.

[61] T. Hoang, A. A. Yavuz, F. B. Durak, and J. Guajardo. Oblivious dynamic searchable encryption on distributed cloud systems. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 113–130. Springer, 2018.

[62] T. Hoang, A. A. Yavuz, and J. Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *CCS*, pages 302–313. ACM, 2016.

[63] Y. Hu, S. Kumar, and R. A. Popa. Ghostor: Toward a secure data-sharing system from decentralized trust. In *NSDI*, pages 851–877, 2020.

[64] Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *Cryptographers' Track at the RSA Conference*, pages 90–107. Springer, 2016.

[65] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12. Citeseer, 2012.

[66] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, volume 3, pages 29–42, 2003.

[67] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 258–274. Springer, 2013.

[68] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *CCS*, pages 965–976. ACM, 2012.

[69] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient byzantine fault tolerance. In *EuroSys*, pages 295–308, 2012.

[70] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *security & Privacy*, pages 895–913. IEEE, 2016.

[71] J. Katz and A. Y. Lindell. Aggregate message authentication codes. In *Cryptographers' Track at the RSA Conference*, pages 155–169, 2008.

[72] G. Kellaris, G. Kollios, K. Nissim, and A. O'neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

[73] Keybase. https://keybase.io/, Accessed 26 May 2020.

[74] A. Kiayias, O. Oksuz, A. Russell, Q. Tang, and B. Wang. Efficient encrypted keyword search for multi-user data sharing. In *ESORICS*, pages 173–195. Springer, 2016.

[75] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Security & Privacy*, pages 880–896. IEEE, 2015.

[76] S. Korokithakis. Writing a full-text search engine using bloom filters, December 2013. https://www.stavros.io/posts/bloom-filter-search-engine/.

[77] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *SOSP*, 41(6):45–58, 2007.

[78] M. Kuzu, M. S. Islam, and M. Kantarcioglu. Efficient similarity search over encrypted data. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1156–1167. IEEE, 2012.

[79] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[80] B. Lampson and D. B. Lomet. A new presumed commit optimization for two phase commit. In *VLDB*, volume 93, pages 630–640, 1993.

[81] A. Langley, E. Kasper, and B. Laurie. Certificate transparency. *Internet Engineering Task Force*, 2013. https://tools.ietf.org/html/rfc6962.

[82] J. Li, M. N. Krohn, D. Mazieres, and D. E. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, volume 4, pages 9–9, 2004.

[83] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. GhostRider: A hardware-software system for memory trace oblivious computation. *ASPLOS*, 50(4):87–101, 2015.

[84] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[85] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500, 2016.

[86] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb. Fast and secure global payments with stellar. In *SOSP*, pages 80–96, 2019.

[87] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*, pages 199–213, 2013.

[88] T. Lovell. Swedish healthcare advice line stored 2.7 million patient phone calls on unprotected web server, February 20 2019. https://www.healthcareitnews.com/news/swedish-healthcare-advice-line-stored-27-million-patient-phone-calls-unprotected-web-server.

[89] S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396. Springer, 2013.

[90] E. MacBrough. Cobalt: BFT governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

[91] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Privacy and access control for outsourced personal records. In *Security & Privacy*, pages 341–358. IEEE, 2015.

[92] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Maliciously secure multi-client oram. In *ACNS*, pages 645–664. Springer, 2017.

[93] T. Mayberry, E.-O. Blass, and G. Noubir. Multi-User Oblivious RAM Secure Against Malicious Servers. *IACR Cryptology ePrint Archive*, 2015:121, 2015.

[94] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *Security & Privacy*, pages 279–296. IEEE, 2018.

[95] E. Nakashima. Russian government hackers penetrated DNC, stole opposition research on Trump, June 14 2016. https://www.washingtonpost.com/world/national-security/russian-government-hackers-penetrated-dnc-stole-opposition-research-on-trump/2016/06/14/cf006cb4-316e-11e6-8ff7-7b6c1998b7a0_story.html.

[96] M. Naveed. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. *IACR Cryptology ePrint Archive*, 2015:668, 2015.

[97] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *Security & Privacy*, pages 639–654. IEEE, 2014.

[98] C. Osborne. Fortune 500 company leaked 264gb of client, payment data, June 7 2019. https://www.zdnet.com/article/veteran-fortune-500-company-leaked-264gb-in-client-payment-data/.

[99] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523. ACM, 1990.

[100] V. Pappas, M. Raykova, B. Vo, S. M. Bellovin, and T. Malkin. Private search in the real world. In *ACSAC*, pages 83–92, 2011.

[101] https://github.com/aricrocuta/oram2pc, Accessed 14 April 2020.

[102] R. Poddar, S. Wang, J. Lu, and R. A. Popa. Practical volume-based attacks on encrypted databases. 2020.

[103] G. S. Poh, J.-J. Chin, W.-C. Yau, K.-K. R. Choo, and M. S. Mohamad. Searchable symmetric encryption: designs and challenges. *ACM Computing Surveys (CSUR)*, 50(3):1–37, 2017.

[104] R. A. Popa and N. Zeldovich. Multi-key searchable encryption. *IACR Cryptology ePrint Archive*, 2013:508, 2013.

[105] D. Porto, J. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth fault tolerance. In *EuroSys*, pages 1–14, 2015.

[106] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *CCS*, pages 1341–1352, 2016.

[107] Preveil. https://www.preveil.com/, Accessed 26 May 2020.

[108] M. Raykova, B. Vo, S. M. Bellovin, and T. Malkin. Secure anonymous database search. In *Workshop on Cloud computing security*, pages 115–126, 2009.

[109] C. Reichert. Payroll data for 29,000 facebook employees stolen, December 13 2019. https://www.cnet.com/news/payroll-data-of-29000-facebook-employees-reportedly-stolen/.

[110] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *FAST*, volume 2, pages 15–30, 2002.

[111] P. Rizomiliotis and S. Gritzalis. ORAM based forward privacy preserving dynamic searchable symmetric encryption schemes. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, pages 65–76. ACM, 2015.

[112] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *Security & Privacy*, pages 198–217. IEEE, 2016.

[113] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. *IACR ePrint*, 2017:549, 2017.

[114] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Security & Privacy*, pages 44–55. IEEE, 2000.

[115] Spideroak. https://spideroak.com/, Accessed 26 May 2020.

[116] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, volume 71, pages 72–75, 2014.

[117] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *CCS*, pages 247–258. ACM, 2013.

[118] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Security & Privacy*, pages 253–267. IEEE, 2013.

[119] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310. ACM, 2013.

[120] H. Sun and S. A. Jafar. The capacity of robust private information retrieval with colluding databases. *IEEE Transactions on Information Theory*, 64(4):2361–2370, 2017.

[121] Sync. https://www.sync.com/, Accessed 26 May 2020.

[122] Q. Tang. Nothing is for free: security in searching shared and encrypted data. *Transactions on Information Forensics and Security*, 9(11):1943–1952, 2014.

[123] Tiny AES in C. https://github.com/kokke/tiny-AES-c, Accessed 24 May 2020.

[124] Tresorit. https://tresorit.com/, Accessed 26 May 2020.

[125] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical private queries on public data. In *NSDI*, pages 299–313, 2017.

[126] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the sixth conference on Computer systems*, pages 123–138, 2011.

[127] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, pages 253–267, 2003.

[128] E. Yuan. Zoom acquires keybase and announces goal of developing the most broadly used enterprise end-to-end encryption offering, May 7 2020. https://blog.zoom.us/wordpress/2020/05/07/zoom-acquires-keybase-and-announces-goal-of-developing-the-most-broadly-used-enterprise-end-to-end-encryption-offering/.

[129] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, pages 707–720, 2016.

[130] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.

A Artifact Appendix

A.1 Abstract

Our DORY prototype is an encrypted search system that splits trust between multiple servers in order to efficiently hide search access patterns from a malicious attacker that controls all but one of the servers. We support parallelism across multiple servers in order to reduce search latency and increase throughput. DORY is written in a combination of C (for the distributed point function and other low-level cryptographic primitives) and Go (for the networking and consensus) for approximately 5,000 lines of code. Our experiment scripts use AWS EC2 instances. Our artifact is available here:

https://github.com/ucbrise/dory

A.2 Artifact check-list

- **Data set:** Enron email dataset used to choose system parameters and set sample documents.
- **Metrics:** Latency, throughput
- **Experiments:** Search latency breakdown, search latency with parallelism, search throughput with parallelism
- **Required disk space:** 18MB
- **Expected experiment run time:** Approximately 4 hours
- **Public link:** https://github.com/ucbrise/dory
- **Code licenses:** Apache v2

A.3 Description

A.3.1 How to access

Our Amazon AWS AMI is public (the AMI IDs for different regions are set in our scripts). See Appendix A.4 for instructions on running scripts for configuring security groups and the key pair as well as starting a cluster.

A.3.2 Software dependencies

We use the hashicorp msgpack library (https://github.com/hashicorp/go-msgpack) for parsing messages and libstemmer (http://snowball.tartarus.org/download.html) for stemming keywords. We build on the DPF implementation in Express [43] (https://github.com/SabaEskandarian/Express). We also use the OpenSSL library for low-level cryptographic primitives.

A.3.3 Data sets

The Bloom filter size in our experiments is based on statistics from the Enron email dataset [30] (see Table 7a). The sample documents to interactively search over in `sample_docs/` are also from the Enron email dataset.

A.4 Installation

The instructions for setting up the Amazon AWS security groups and key pair are available here: https://github.com/ucbrise/dory#setting-up-aws-security-groups-and-keypairs. The instructions for starting a cluster of EC2 instances using our public AMIs are available here: https://github.com/ucbrise/dory#setup. We use `r5n.4xlarge` instances in different regions that are geographically close (`east-1` and `east-2`). We also provide instructions for building from source here: https://github.com/ucbrise/dory#building-from-source.

A.5 Experiment workflow

To start running experiments, the reviewer should first create a cluster (Appendix A.4). Each figure (or group of figures) reproduced has a corresponding script to run the experiment. Each figure reproduced has another script to plot the data collected. Details are available here: https://github.com/ucbrise/dory#running-experiments. After running experiments, the reviewer should teardown the cluster following instructions here: https://github.com/ucbrise/dory#setup.

Because the ORAM baseline experiments in our paper take approximately a week to run, we only reproduce two data points (1,024 and 2,048 documents), making the experiment take a little over an hour.

A.6 Evaluation and expected result

The above instructions reproduce Table 7b, Figure 8, Section 7.2, Figure 10, and Figure 11. There may be some variation from the figures in the paper based on how long the experiments are allowed to run.

Our scripts plot the figures using the ORAM baseline data we collected ourselves, as the experiments we provide for reviewers only reproduce two data points. Reviewers can

compare the two data points we reproduce to the data we collected to verify that the data matches up.

More detailed instructions on running experiments and interpreting results are available here: https://github.com/ucbrise/dory#running-experiments.

A.7 Experiment customization

Reviewers can configure experiments to run for more trials, run for different numbers of documents, or use different Bloom filter sizes.

A.8 Notes

We implement the DORY search protocol as described in the body of the paper, and our implementation does not include a complementary end-to-end encrypted filesystem that could use or interface with DORY. We support keyword search with a small, configurable number of false positives (we do not support regular expressions or other advanced search features).

A.9 AE Methodology

Submission, reviewing and badging methodology:

https://www.usenix.org/conference/osdi20/call-for-artifacts



SafetyPin: Encrypted Backups with Human-Memorable Secrets

Emma Dauterman
UC Berkeley

Henry Corrigan-Gibbs
EPFL and MIT CSAIL

David Mazières
Stanford

Abstract. We present the design and implementation of SafetyPin, a system for encrypted mobile-device backups. Like existing cloud-based mobile-backup systems, including those of Apple and Google, SafetyPin requires users to remember only a short PIN and defends against brute-force PIN-guessing attacks using hardware security protections. Unlike today's systems, SafetyPin splits trust over a cluster of hardware security modules (HSMs) in order to provide security guarantees that scale with the number of HSMs. In this way, SafetyPin protects backed-up user data even against an attacker that can adaptively compromise many of the system's constituent HSMs. SafetyPin provides this protection without sacrificing scalability or fault tolerance. Decentralizing trust while respecting the resource limits of today's HSMs requires a synthesis of systems-design principles and cryptographic tools. We evaluate SafetyPin on a cluster of 100 low-cost HSMs and show that a SafetyPin-protected recovery takes 1.01 seconds. To process 1B recoveries a year, we estimate that a SafetyPin deployment would need 3,100 low-cost HSMs.

1 Introduction

Modern mobile phones and tablets back up sensitive data to the cloud. To protect users' privacy, this data must be encrypted under keys that are not available to the cloud provider. Unfortunately, with 3.8 billion smartphone users, it is impractical to expect them all to store, say, a 128-bit AES backup key. Not everyone has a computer, or trustworthy friends who can keep shares of a backup key, or even a safe place to store a backup key on paper. As a result, mobile OSes have fallen back to protecting backups with the least common denominator: device screen-lock PINs. Using PINs is good for security because a user's screen-lock PIN never leaves her device (so the cloud provider never learns it). Using PINs is good for usability because users generally remember them.

Unfortunately, PINs have such low entropy (e.g, six decimal digits) that no feasible amount of key stretching can protect against brute-force PIN-guessing attacks. Instead, modern backup systems—such as those from Apple [47], Google [82], and Signal [55]—rely on hardware-security modules (HSMs) in their data centers to thwart brute-force attacks. Specifically, devices encrypt their backup keys under the public keys of HSMs, but each device includes a hash of its screen-lock PIN as part of the plaintext. HSMs return decrypted plaintext only to clients that can supply this PIN hash. Furthermore, HSMs limit the number of decryption attempts for any given user account. For fault tolerance, a device typically encrypts its backup key to the public keys of five HSMs, allowing any one of the five to recover the backup key.

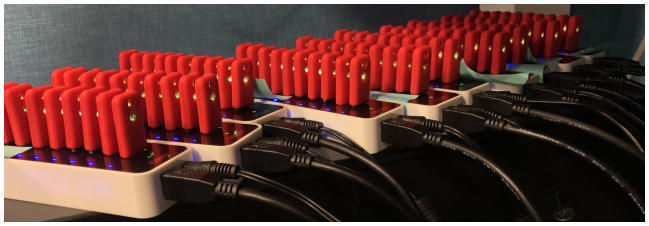


Figure 1: Our cluster of 100 low-cost hardware security modules (SoloKeys [72]) on which we evaluate SafetyPin.

This status quo still falls short of acceptable privacy for two reasons. First, HSMs are not perfect, yet each HSM in these systems is a single point of security failure for millions of users' backup keys. Second, these systems make it difficult for clients to detect security breaches. For instance, if a malicious insider working in a data center physically steals an HSM, then to anyone outside the company it looks like an unremarkable single hardware failure. Alternatively, if an insider successfully guesses someone's PIN, the victim may have no idea her backup was ever compromised.

This paper presents SafetyPin, a PIN-based encrypted-backup system with stronger security properties. The key idea behind SafetyPin is that recovering any user's backed-up data either requires (a) guessing the user's PIN or (b) compromising a very large number of HSMs—e.g., 6% of all HSMs operated by a provider. (The 6% figure here is a tunable system parameter.) Such large-scale attacks would typically need to span multiple data centers, be harder for insiders to pull off undetected against physical devices, cost more, and also likely cause service disruptions visible to end users.

One way to achieve SafetyPin's security goal would be to threshold-encrypt the client's hashed PIN and backup key in such a way that decrypting the client's backup key would require the participation of 6% of all HSMs in the system. Unfortunately, this approach lacks scalability. If each client recovering a backup must interact with 6% of the system's HSMs, adding more HSMs improves security without improving throughput. As the number of HSMs in the system increases, we would like the system's overall throughput to increase in tandem with its security (i.e., the attacker's cost).

To achieve scalability, SafetyPin takes a different approach:

devices threshold-encrypt their backup keys to a small cluster of n HSMs such that decryption requires the participation of most HSMs in the cluster. The cluster size n is independent of the total number of HSMs in the system, and depends on both the fraction of compromised HSMs the system can tolerate and the fraction of HSMs that can fail-stop. (For example, to tolerate the compromise of 6% of HSMs where half of a cluster is allowed to fail-stop, we can set the cluster size $n = 40$.) This design achieves our scalability goal, since each device need only communicate with a small fixed number of HSMs during recovery. This design also achieves our security goal because the cluster of n HSMs that can decrypt a client's backup depends on the client's secret PIN, via a primitive we introduce called *location-hiding encryption*. Hence, even if an attacker compromises 6% of the HSMs in the system as a whole, the chances that the attacker compromises a "useful" set of HSMs—i.e., at least half of the HSMs in the device's chosen cluster—is very small. More precisely, we show that if the total number of HSMs in the system is large enough (a few hundred or more), the probability that an attacker can decrypt a backup via HSM compromise is not much higher than the probability of simply guessing the client's PIN.

In modern backup systems, each HSM only needs to monitor the number of PIN attempts for a small subset of users, but because of our location-hiding encryption primitive, every HSM needs to be able to verify the number of PIN attempts for every user. To maintain this information scalably, the HSMs use a new type of distributed log. Third parties can monitor this log to alert users whenever a backup-recovery attempt is underway. Since a compromised service provider may see which HSMs a mobile device interacts with during recovery (and could compromise those HSMs to recover the users' backed-up data), HSMs revoke their ability to decrypt backups after completing the recovery process. Implementing this revocation requires adapting "puncturable encryption" [38] to storage-limited HSMs. While our prototype is focused on PIN-protected backups, these primitives have potentially broader applicability to problems such as private storage in peer-to-peer systems and cryptocurrency "brain wallets."

We implemented SafetyPin on low-cost SoloKey HSMs [72]. We evaluate the system using a cluster of 100 SoloKeys (Figure 1) and an Android phone (representing the client device). Generating a recovery ciphertext on the client, excluding the time to encrypt the disk image, takes 0.37 seconds. To process 1B recoveries a year, or 123K recoveries per hour, we estimate that we would need 3,100 SoloKeys. In a SafetyPin deployment of 3,100 HSMs, tolerating the compromise of 6% of the HSMs (i.e., 194 HSMs), the client must interact with a cluster of 40 HSMs during recovery. Running our backup-recovery protocol across a cluster of this size takes 1.01 seconds.

Limitations. A limitation of SafetyPin is that the set of HSMs a device uses for recovery can leak information about the user's PIN. In particular, an attacker who controls the data center can learn a salted hash of the user's PIN during recovery. This is unfortunate in the common case that people re-use the same PIN after recovery [23, 42, 32, 70]. We discuss one mitigation in Section 8. Also, while it is possible to detect when PINs can safely be re-used, we have not yet implemented this functionality.

In addition, SafetyPin is more expensive than today's PIN-based backup systems. SafetyPin requires the data center operator to operate a much larger fleet of HSMs (roughly 50 – 100× larger) than the standard HSM-based backup systems require. SafetyPin clients must also download roughly 2MB of keying material per day in a SafetyPin deployment supporting one billion recoveries per year, due to the periodic rotation of large HSM keys. Even so, we expect that the cost of storing and transferring disk images (GBs/user) will dwarf these costs.

2 The setting

Entities. Our encrypted-backup system involves three entities, whose roles we describe here.

Client. Initially, the client holds (1) a username with the service provider, (2) a human-memorable passphrase or PIN, (3) a disk image to be backed up, and (4) the public keys of the service provider's HSMs. Later on, the client should be able to recover her backed-up data using only her username, her PIN, and access to the other components of the backup system.

In SafetyPin, as in today's PIN-based backup systems, security depends on the client having access to the HSMs' true public keys: If a malicious service provider can swap out the HSMs' true public keys for its own public keys without detection, the service provider can immediately break security. Using a distributed log (Section 6) can ensure that all clients see a common set of HSM public keys, to prevent targeted attacks. Hardware-attestation techniques, as used in the FIDO [67] and SGX [44] specs, can provide another defense.

We also assume the provider has traditional account authentication (e.g., Gmail passwords) to prevent random third parties from consuming PIN guesses, but we omit this from the discussion for simplicity.

Service provider. The service provider offers the encrypted-backup service to a pool of clients and it maintains the data centers in which the backup system runs. For example, the service provider could be a mobile-phone vendor, such as Apple or Google. The service provider's data centers contain the network infrastructure that connects the HSMs. They also contain large amounts of (potentially untrustworthy) storage and computing resources. Our security properties will hold against a service provider that becomes compromised at any point after the system is set up.

Hardware security modules (HSMs). The service provider's data centers contain thousands of hardware security modules. An HSM is a tamper-resistant computing device meant for storing cryptographic secrets. HSMs have fully programmable processors but are typically resource-poor (see Table 2). It is possible to lock an HSM's firmware before deployment,

Device	Price	g^x/sec.	Storage	FIPS
SoloKey [72]	$20	8	256 KB⋆	
YubiHSM 2 [84]	$650	14	126 KB	
SafeNet A700 [68]	$18,468	2,000	2,048 KB	✓
Intel i7-8569U (CPU)	$431	22,338	n/a	

Table 2: Hardware security modules offer physical security protections but are computationally weak compared to a standard CPU.

The g^x/sec is NIST P256 elliptic-curve point-multiplications per second. "FIPS?" refers to whether the device meets the FIPS 140-2 standard for HSMs. (⋆ The 256 KB storage on the SoloKey is shared between code and data.)

which makes remote compromise and key-extraction attacks more difficult. Each HSM has a public key and stores the corresponding secret key in its secure memory.

The attack scenario. The service provider (Apple, Google, etc.) spends vast amounts of money acquiring a large user base for products that store user data in the cloud. The provider risks reputational damage and journalistic scrutiny if it cannot ensure the durability and confidentiality of user data.

A service provider can deploy SafetyPin as a way to build trust among its user base and to protect its own infrastructure against future compromise. By enlisting third-party organizations to monitor the SafetyPin deployment's public distributed log, the provider can build further public trust in the system.

At some point after the provider deploys SafetyPin, a powerful attacker wishes to steal user data. The attacker may have malicious insiders working for the provider. It may physically compromise data centers to steal HSMs. It may intercept shipments to tamper with some of the HSMs on their way to the data center. The attacker could also be a state actor employing legal pressure to gain access to data centers. Nonetheless, the attacker is sensitive to both the cost of attacks and the risk of public exposure.

Both the attack cost and risk of exposure increase with the number of HSMs the attacker must compromise. For instance, while a malicious insider working at a data center may be able to abscond with a single HSM—passing the missing device off as a hardware failure—removing 100 HSMs is a much riskier proposition. A state actor who can order the provider to hand over HSMs may be dissuaded if doing so will attract press coverage either by making non-targeted clients' data unrecoverable or creating a damning public audit trail.

The attacker may compromise clients as well as the provider. For instance, the attacker may have a good guess at a target user's PIN, perhaps because of CCTV footage showing the user unlocking a mobile device. While SafetyPin cannot prevent the attacker from gaining access to the data with the correct PIN, the risk will be higher to the attacker if stolen PINs cannot be used without exposing the attack in SafetyPin's public distributed log.

Notation. The set $\mathbb{Z}^{>0}$ refers to the set of natural numbers $\{1, 2, 3, \dots\}$. For a positive integer n, we let $[n] = \{1, \dots, n\}$ and we use $\perp$ to denote a failure symbol. For strings a and b, we write their concatenation as $a \| b$. Throughout, we use λ to denote the security parameter, and we typically take $\lambda = 128$ (i.e., for 128-bit security).

3 System goals

SafetyPin implements an encrypted-backup functionality, which consists of two routines:

- the *backup* algorithm, which the client uses to produce its encrypted backup, and
- the *recovery* protocol, in which the client uses HSMs to recover the backup plaintext from ciphertext.

We define these protocols with respect to a number of HSMs $N \in \mathbb{Z}^{>0}$ and a finite PIN space $\mathcal{P} \subseteq \{0,1\}^*$. For convenience, we define the *master public key* mpk for a data center to be all N HSMs' public keys: $\mathsf{mpk} = (\mathsf{pk}_1, \dots, \mathsf{pk}_N)$. The syntax of an encrypted-backup system is then as follows:

$\mathsf{Backup}(\mathsf{mpk}, \mathsf{user}, \mathsf{pin}, \mathsf{msg}) \to \mathsf{ct}$. Given the master public key mpk, a client username user, the client's PIN $\mathsf{pin} \in \mathcal{P}$, and a message $\mathsf{msg} \in \{0,1\}^*$ to be backed up, output a recovery ciphertext ct. This routine runs on the client and requires no interaction with HSMs. The client uploads the resulting ciphertext ct to the service provider.

$\mathsf{Recover}^{\mathcal{S}, \mathcal{H}_1, \dots, \mathcal{H}_N}(\mathsf{mpk}, \mathsf{user}, \mathsf{pin}, \mathsf{ct}) \to \mathsf{msg}$ or $\perp$. The client initiates the recovery routine, which takes as input the master public key mpk, a client username user, a PIN $\mathsf{pin} \in \mathcal{P}$, and a recovery ciphertext ct.

During the execution of $\mathsf{Recover}$, the client interacts with the service provider $\mathcal{S}$ and a subset of the HSMs $\mathcal{H}_1, \dots, \mathcal{H}_N$. Each HSM $\mathcal{H}_i$ holds the master public key mpk, and its secret decryption key sk_i. During recovery, the data center provides the client's username user to each HSM.

The recovery routine outputs a backed-up message $\mathsf{msg} \in \{0,1\}^*$ or a failure symbol $\perp$.

We now describe the security properties that such a system should satisfy. We work in an asynchronous network model; we use standard cryptographic primitives to set up authenticated and encrypted channels between the client, service provider, and HSMs.

Property 1: Security. If the client obtains the HSMs' true public keys, then even an attacker that:

- controls the service provider (in particular, is an active network attacker inside the data centers and has control of the service provider's servers and storage),
- compromises an f_{secret} (e.g., $f_{\mathsf{secret}} = \frac{1}{16}$) fraction of HSMs in the data center *before* the client begins the recovery process, and
- compromises all of the HSMs in the data center *after* the recovery protocol completes,

still should learn nothing about any honest client's encrypted message (in a semantic-security sense [35]) beyond what it

can learn by guessing that client's PIN.

Discussion: The adversary can inspect all clients' recovery ciphertexts and then choose to compromise a large set of HSMs that depends on these ciphertexts. Such attacks are relevant when, for example, a state actor with the power to compromise many HSMs targets the backed-up data of a specific set of users.

Two important caveats are: (1) SafetyPin does not protect against an attacker compromising HSMs while recovery is in progress (see Figure 4) and (2) as implemented, SafetyPin does not protect the PIN: an adversary that observes which HSMs the client contacts during recovery may learn a salted hash of the PIN after recovery completes. Section 6.3 discusses how to detect and mitigate this leakage by protecting the salt.

Property 2: Scalability. The recovery protocol should require the client to interact with a constant number of HSMs, independent of the number of HSMs in the data center. (This constant may depend on the security parameter and on the fraction of HSMs whose compromise the system can tolerate.) Hence, providers can deploy additional HSMs to scale capacity. Concretely, when we configure the system to tolerate the compromise of $f_{\mathsf{secret}} = \frac{1}{16}$ of the data center's HSMs, our protocol requires the client to communicate with 40 HSMs during recovery.

Property 3: Fault tolerance. Every client should be able to recover her encrypted message even if a constant fraction f_{live} (e.g. $f_{\mathsf{live}} = \frac{1}{64}$) of the HSMs in the data center fail-stop.

Setting parameters. For the remainder of this paper, we set the fraction of compromised HSMs that the system can tolerate to $f_{\mathsf{secret}} = \frac{1}{16}$ and the fraction of HSMs that can fail while still allowing the client to recover her backup to $f_{\mathsf{live}} = \frac{1}{64}$. This choice is reasonable because large companies have more than 16 data centers, while smaller companies can collaborate on a shared deployment with 16 physical security perimeters. By adjusting the other parameters, it is possible to achieve any $0 < f_{\mathsf{secret}} < 1$ or $0 < f_{\mathsf{live}} < 1$. (In Section 9.2, we discuss how the choice of these values affects other system parameters.)

4 Architecture overview

We now describe our encrypted-backup protocol (Figure 3) and explain how it satisfies the design goals of Section 3. We will discuss possible extensions and deployment considerations in Section 8.

4.1 The back-up process

The client begins the back-up process holding

- the public keys of all HSMs in the data center,
- its secret PIN, and
- a disk image to be backed up (the "message").

To back up its disk image, the client samples a subset of n HSMs out of the N total HSMs in the data center where $n \ll N$. The client chooses this subset by hashing (a) public information: the service name, its username, and a public salt the client chooses at random, and (b) its secret PIN. The client then encrypts its message with a random AES encryption key, and then splits this AES key into n threshold shares using Shamir secret sharing [69], such that any threshold t of the shares suffice to recover the AES key. The client prepends each share with the client's username to ensure that the ciphertexts are bound to the client's username. The client then encrypts one share to the public key of each HSM in its chosen subset.

The client's recovery ciphertext then consists of: its public salt, the AES-encrypted message, the n encrypted shares of the AES key, and a configuration-epoch number that the service provider can use to identify the set of HSMs that were in service at the time the client created its backup. The client computes the ciphertext locally and uploads it to the backup service provider, with no HSM interactions required.

To explain why this construction is scalable: since only a constant number of HSMs $n \ll N$ participate in the decryption process, the system scales well as the number of HSMs in the data center increases.

To explain why this construction should be secure: if the attacker cannot guess the client's PIN, the attacker does not know which set of n HSMs (out of the N total) it needs to compromise to recover the client's AES key. So, the best attacks are either to: guess the client's PIN or compromise a large fraction of the data center.

This argument requires that each individual key-share ciphertext leak no information about which HSM can decrypt it—a cryptographic property known as "key privacy" [8]. However, even key-private encryption schemes do not always remain secure against an adversary that adaptively compromises secret keys, which leads to our first technical challenge:

Challenge 1. *How can we ensure that the client's recovery ciphertext "leaks nothing" about which HSMs are required to decrypt the client's message, even against an attacker who can adaptively compromise HSMs?*

In Section 5, we explain how to solve this problem using *location-hiding encryption*, a new cryptographic primitive.

4.2 The recovery process

The client begins the recovery process holding:

- the public keys of all HSMs in the data center,
- its secret PIN, and
- its recovery ciphertext (which the client can fetch from the service provider).

First, the client asks the service provider to record its recovery attempt in the *append-only log*, implemented collectively by the service provider and HSMs. The log holds a mapping of identifiers to values. The service provider can insert new identifier-value pairs into the log but the service provider cannot modify or delete the values of defined identifiers, ensuring that there is at most one immutable value for each identifier.

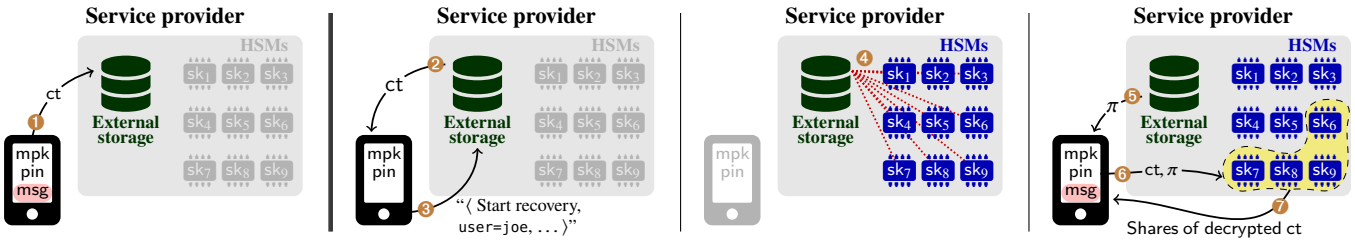


Figure 3: An overview of the recovery-protocol flow. Each HSM i holds a secret key sk_i. The client holds a vector mpk of all HSMs' public keys. ❶ During backup, the client uses its PIN and the master public key to encrypt its data msg into a recovery ciphertext ct. The client then uploads this recovery ciphertext ct to the service provider. ❷ During recovery, the client downloads its recovery ciphertext. ❸ The client asks the data center to log its recovery attempt. ❹ The service provider collects a batch of client log-insertion requests, updates the log, and aggregates the new log into a Merkle tree. The service provider and HSMs run a log-update protocol. At the end of this protocol, each HSM holds the root of the Merkle tree computed over the latest log. ❺ The service provider sends the client a Merkle proof π that the client's recovery attempt is included in the latest log (i.e., in the latest Merkle root). ❻ The client sends the recovery ciphertext ct and log-inclusion proof π to the subset of HSMs needed to decrypt the recovery ciphertext. ❼ The HSMs check the proof and return shares of the decrypted ciphertext to the client. The client uses these to recover the backed-up data msg.

The recovery attempt is logged as follows. The client begins by using public information (service name, username, and salt in the recovery ciphertext) along with its secret PIN to recover the subset of n HSMs it picked during backup. The client then hashes these values together with some randomness to produce a cryptographic commitment h to the identities of these HSMs and to its recovery ciphertext. The client then asks the service provider to insert the identifier-value pair (user, h) into the log, where user is the client's username. (In this discussion, we use the client's username as the key for simplicity. In practice, to preserve privacy, we might use an opaque device-install UUID.)

The service provider collects a batch of these log-insertion requests, produces a Merkle-tree [59] digest over the updated log, and runs a log-update protocol with the HSMs. At the end of this protocol, the HSMs hold the updated log digest. The service provider then returns to the client a Merkle proof π proving that the pair (user, h) appears in the latest log digest.

Since the service provider and HSMs run the log-update protocol periodically (e.g., every 10 minutes), the client will have to wait a few minutes on average to decrypt its backup. The client already has to download its large encrypted disk image, which will likely take minutes, so these steps can proceed in parallel.

The client then contacts its chosen set of n HSMs over an encrypted channel, such as TLS. The client sends to each HSM: its username, the opening of its commitment h (i.e., the values and randomness used to construct the commitment h), and the Merkle inclusion proof π. Each HSM

- recomputes the commitment h and checks the inclusion proof π (to confirm that the recovery attempt is logged), and
- decrypts its share of the client's AES key, confirms that the username in the decrypted plaintext matches the one provided by the client (which prevents user A from attempting to decrypt user B's ciphertext, in collusion with a malicious service provider).

If both of these checks pass, the HSM returns the AES-key share to the client.

Given any t of these decryption-key shares, the client can recover the AES key used to encrypt its backup. The client can then use this AES key to decrypt its backed-up message.

Since at most one log entry can exist per username, the use of the log ensures that each user can make at most one recovery attempt. In this way, the system defeats brute-force PIN-guessing attacks. With a slight modification, it is possible to allow each user to make a fixed number (e.g., 5) guesses, or a fixed number of guesses per time period (e.g., 5 per month).

A counter-intuitive property of this scheme is that the client never explicitly provides its PIN to the HSMs. The fact that the client knows which subset of the HSMs to contact implicitly proves the client's knowledge of the PIN because the set of n HSMs is much smaller than the total number of HSMs N.

This overview leaves some technical details unexplained. In particular:

Challenge 2. *How do the HSMs implement the append-only log without sacrificing scalability or security?*

A straightforward way to implement the log would be to have each HSM store the entire state of the log. But then every HSM would have to participate in every recovery attempt, which would not meet our scalability goals. Another implementation would be to have the data-center operators maintain the log, but then malicious data centers could violate the append-only property, and thus mount brute-force PIN-guessing attacks, without HSMs noticing.

In Section 6, we explain how the HSMs can collectively maintain such an append-only log in a scalable and secure manner. At a high level, the (potentially adversarial) data center maintains the state of the log, which we represent as a list of identifier-value pairs. Every time the data center wants to insert an identifier-value pair into the log, the data center must

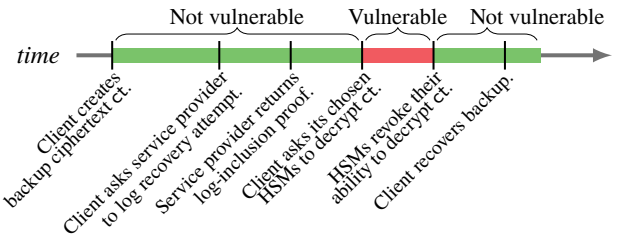


Figure 4: Since HSMs in SafetyPin revoke their ability to decrypt a client's recovery ciphertext, SafetyPin protects against HSM compromise attacks that take place before recovery begins and after it completes. An attacker who can compromise HSMs while recovery is in progress can break security.

prove to a random subset of the HSMs that the identifier to be inserted is undefined in the current log. Provided that at least one honest HSM audits each log-insertion, we can guarantee that the values associated with log identifiers are immutable (i.e., that we maintain the log's append-only property). In this way, (a) each HSM needs to participate in only a vanishing fraction of the recovery attempts and (b) even an attacker who can compromise many of the HSMs cannot break the append-only nature of the log.

One remaining issue is that an attacker who observes the data center network may see which HSMs a client interacts with during recovery and decide to compromise that exact set of HSMs after recovery completes.

Challenge 3. *For scalability, the client should only communicate with a small number of HSMs during recovery. But then how can we protect against an attacker who compromises these HSMs after recovery completes?*

Our idea is as follows: after a client runs the recovery protocol, each participating HSM *revokes* its ability to decrypt that client's recovery ciphertext. So, even if an after-the-fact attacker compromises the HSMs that participated in recovery, the attacker learns no useful information. The only window of vulnerability is at the moment after the client contacts its HSMs and before the HSMs complete revocation (Figure 4). Making this work on resource-limited HSMs requires new technical tools, which we describe in Section 7.

5 Protecting the mapping of users to HSMs with location-hiding encryption

In this section, we define and construct *location-hiding encryption*, which the client uses to encrypt its backup data.

The location-hiding encryption routine takes as input (1) a set of N public keys, (2) a short PIN, and (3) a message, and outputs a ciphertext. In our application, the N public keys are the public keys of the N HSMs in the data center.

The cryptosystem has three main properties, which we formalize in the full version [24]:

1. Security. To successfully decrypt the ciphertext, an attacker must either (a) guess the PIN or (b) control more than a constant fraction f_{secret} of the N total secret keys. This security property must hold even if the attacker can adaptively compromise an f_{secret} fraction of the N secret keys. In our application, this implies that unless an adversary can guess the PIN or compromise a constant f_{secret} fraction of the HSMs in the data center, it learns nothing about the client's backed-up data.

2. Scalability. Given the PIN used to encrypt the message, it is possible to decrypt the message using a small subset of the N secret keys corresponding to the N public keys used during encryption. In our application, a client who knows the correct PIN can recover its backup by interacting with only a small cluster of n HSMs (for some parameter $n \ll N$) out of the N total HSMs, So as N grows, each HSM needs to participate in a vanishing fraction of the total recovery attempts.

3. Fault tolerance. Given the PIN, it is possible to decrypt a ciphertext even if a random fraction f_{live} of all secret keys are unavailable. In our application, this implies clients can recover their backups even if an f_{live} fraction of all HSMs fail.

We call this primitive "location-hiding encryption" because there is a small set of n HSMs that the attacker could compromise to decrypt the ciphertext, but the cryptosystem *hides the location* of these HSMs within the larger pool of N HSMs.

Our construction

Our construction of location-hiding encryption is just a careful composition of existing primitives. However, it takes some analysis to prove that the composition provides the desired security properties. We describe our construction here in prose and we include the security definitions and proofs in the full version [24]. The construction makes use of a public-key encryption scheme (hashed ElGamal encryption [27, 15]) and an authenticated encryption scheme (e.g., AES-GCM).

Setup. In our construction, each HSM i, for $i \in [N]$, holds a keypair $(\mathsf{pk}_i, \mathsf{sk}_i)$ for the public-key encryption scheme. Let $t \in \mathbb{Z}^{>0}$ be a threshold such that if each HSM fails with probability f_{live}, then in a random sample of n HSMs, there are at least t non-failed HSMs with extremely high probability. Our instantiation takes $t = n/2$ for $f_{\mathsf{live}} = \frac{1}{64}$.

Encryption. The encryption routine takes as input a list of N public keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_N)$, a PIN, and a message msg. To encrypt the message using our location-hiding encryption scheme:

1. Sample a random AES key k and a random salt.
2. Split k into t-out-of-n-Shamir secret shares $k_1, \ldots, k_n$ [69].
3. Hash the PIN and salt and use the result as a seed to generate a list of n random indices $i_1, \ldots, i_n \in [N]$.
4. Encrypt each key-share k_j with public key pk_{i_j}.
5. Finally, return (a) the salt, (b) the n public-key ciphertexts, and (c) the AES encryption of msg under key k.

Decryption. To decrypt given the ciphertext and PIN:

1. Hash the salt and PIN to reconstruct the set of indices $i_1, \ldots, i_n \in [N]$ used during encryption.
2. Use secret keys $\mathsf{sk}_{i_1}, \ldots, \mathsf{sk}_{i_n}$ to decrypt the n shares of the AES key k. (In fact, only t of the shares are necessary.)
3. Using the recovery routine for Shamir secret sharing, recompute the AES key k from its shares.
4. Decrypt and return msg using the AES key k.

Notice that the decryption routine only uses the PIN to sample the set of secret keys used for decryption. In our application, this implies that the client never needs to explicitly provide its PIN (or even a hash of its PIN) to the HSMs; contacting the right subset of HSMs is enough to ensure that the client provided the correct PIN.

The intuition behind the security analysis is straightforward: with hashed ElGamal encryption, the ciphertext reveals no information about which n public keys (out of the N total where $n \ll N$) were used during encryption. Thus, the ciphertext reveals no information about which secret keys the attacker must compromise unless the attacker can guess the PIN. Without these secret keys, the attacker cannot learn anything about k, and therefore cannot decrypt the message.

In the full version [24], we formalize our location-hiding encryption scheme and prove that it is secure in the random oracle model when instantiated with the hashed ElGamal encryption (with certain constraints on n and N).

There are two reasons why the security analysis is non-trivial: First, we must ensure that the ciphertext leaks nothing about the n keys to which it was encrypted (i.e., that it is *key-private* [8]). Second, we must ensure that the encryption scheme remains secure even if an attacker can adaptively compromise secret keys. This is known as security under *selective-opening attack* [9, 29, 43]. Showing that both properties hold at once is the source of the technical complexity.

6 The distributed log

In SafetyPin, the HSMs collectively maintain a *distributed log*, which any external party can read and replay. The service provider maintains the log state and the HSMs monitor log insertions to ensure that the service provider does not violate the log's append-only property.

We use this log for two primary purposes:

1. **Limiting PIN guesses.** To prevent an attacker from brute-force guessing a client's PIN, we use the log (as described in Section 4) to enforce a global limit on the number of recovery attempts that the HSMs allow per username.
2. **Monitoring recovery attempts.** The service provider logs each recovery attempt, so any SafetyPin client can inspect the log to learn whether someone (e.g., a foreign attacker or snooping acquaintance) has tried to recover their backed-up data. A client could then take mitigating action—such as contacting their service provider, a law-enforcement agency, or the press.

A third use for the log—which comes directly from related work [30] and which we have not yet implemented—is to **manage HSM group membership**. Whenever the service provider wants to add or remove an HSM from the data center, the service provider operator could record this information in the log before the other HSMs will accept the change. All SafetyPin clients can thus verify that they are communicating with the same set of HSMs. In addition, clients can also detect suspicious changes in the set of HSMs in the data center. (For example, if the service provider replaces all HSMs in the data center over the course of a day.)

The log is simply a list of identifier-value pairs maintained by the service provider. Clients can insert identifier-value pairs in order to record recovery attempts, and HSMs maintain a digest of the log state. Our distributed log must satisfy the following key property:

> *If any honest HSM ever accepts that an identifier-value pair* $(\mathsf{id}, \mathsf{val})$ *is included in the log, the HSM should never accept that* $(\mathsf{id}, \mathsf{val}')$ *is included in the log, for any value* $\mathsf{val}' \neq \mathsf{val}$.

6.1 Underlying data structure

Terminology. The log L is a list of key-value pairs. Since we use the word "key" in this paper to refer to cryptographic keys, we call log keys "identifiers." We say that a log L' "extends" a log L if (a) L is a prefix of L' and (b) every identifier in L' appears at most once.

Our distributed log uses an authenticated data structure [75, 64, 77] that implements the following five routines:

$\mathsf{Digest}(L) \to d$. Return a constant-size digest d representing the current state of the log.

$\mathsf{ProveIncludes}(L, \mathsf{id}, \mathsf{val}) \to \{\pi_{\mathsf{Inc}}, \perp\}$. Output a proof π_{Inc} that attests to the fact that the identifier-value pair $(\mathsf{id}, \mathsf{val})$ is in the log represented by digest $d = \mathsf{Digest}(L)$.

$\mathsf{DoesInclude}(d, \mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}}) \to \{0, 1\}$. Return "1" iff π_{Inc} proves that the log that digest d represents contains $(\mathsf{id}, \mathsf{val})$.

$\mathsf{ProveExtends}(L, L') \to \{\pi_{\mathsf{Ext}}, \perp\}$. Output a proof π_{Ext} that $d' = \mathsf{Digest}(L')$ represents a log that extends the log that digest $d = \mathsf{Digest}(L)$ represents.

$\mathsf{DoesExtend}(d, d', \pi_{\mathsf{Ext}}) \to \{0, 1\}$. Return "1" iff π_{Ext} proves that the log that digest d' represents extends the log that digest d represents.

The inclusion and extension proofs must be complete (honest verifiers accept valid proofs) and sound (honest verifiers reject invalid proofs), as we define in the full version [24].

Implementing the data structure. Nissim and Naor [64] show that it is possible to implement these log primitives using only Merkle trees [59]. We summarize their construction in the full version [24]. At a very high level: the digest of the log is just the root of a Merkle tree computed over all of the entries of the log, represented as a binary search tree indexed by id. A log-inclusion proof π_{Inc} is a Merkle proof of inclusion relative to this root. A log-extension proof π_{Ext} is a proof that: (1) every identifier inserted to the new log did not exist in the

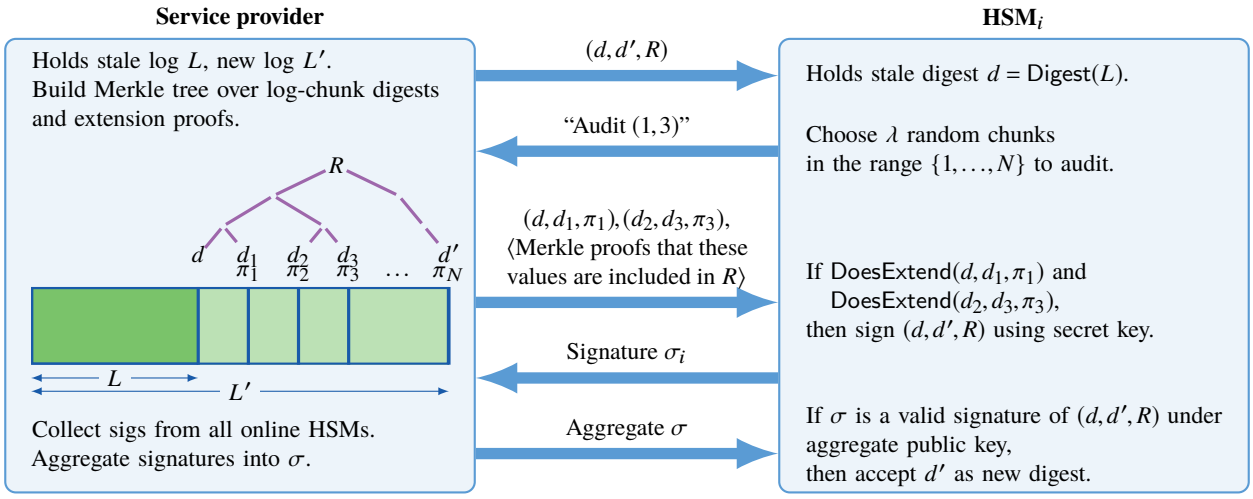


Figure 5: The protocol that the service provider and HSMs use to update the HSM's log digest.

old log and (2) the new digest represents the old log tree with the new values inserted. It is possible to prove both assertions using a number of Merkle proofs proportional to the number of log insertions.

6.2 Building a distributed log

We now explain how to use the primitives of Section 6.1 to build our distributed append-only log.

Initializing the log. The service provider maintains the entire state of the log L. Each HSM stores a log digest d which, in steady state, is the digest of the log L that the service provider holds. Initially, the log L is empty and each HSM holds the digest of the empty log.

Inserting into the log. A client can insert an entry $(\mathsf{id}, \mathsf{val})$ into the log by simply sending the pair to the service provider. The service provider adds this entry to its log state L.

Proving log membership to HSMs. Before the HSMs allow a client to begin the recovery process, the HSMs require proof that the client's recovery attempt is logged. Assume for the moment that the service provider holds a log L and all HSMs hold the up-to-date digest d = Digest(L). (We will explain how the HSMs get the latest log digest in a moment.) Then, a client can prove inclusion of any pair $(\mathsf{id}, \mathsf{val})$ in the log by asking the service provider for an inclusion proof. The service provider computes π_{Inc} = ProveIncludes($L, \mathsf{id}, \mathsf{val}$) and returns the inclusion proof to the client. The client then sends $(\mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}})$ to the HSM, which can check DoesInclude$(d, \mathsf{id}, \mathsf{val}, \pi_{\mathsf{Inc}})$ to be convinced that $(\mathsf{id}, \mathsf{val})$ is in the log represented by its digest d. This inclusion check is fast—logarithmic in the log length.

Updating the log digest at the HSMs. After a sequence of log-insertions, the service provider holds a log state L'. The HSMs will be holding a digest d = Digest(L) of a stale log L. If the service provider is honest, the new log L' extends the old log L.

To update the log digest at the HSMs, the service provider will first send the new digest d' = Digest(L') to every HSM. Next, the data center must convince each HSM that this new digest d' represents a log that extends the log L that the old digest d represents.

One non-scalable way to achieve this would be for the service provider to send an extension proof π_{Ext} = ProveExtends(L, L') to every HSM. The problem is that the time required to check this extension proof grows linearly with the number of new log entries. So if every HSMs checked the entire extension proof, the throughput of the system would not increase as the number of HSMs increases.

Instead, we use a randomized-checking approach, as in Figure 5. If there have been I insertions to the log since the last update, the service provider divides the updates into N chunks, each containing I/N insertions. The service provider then applies these chunks of updates to the old log L one at a time, producing a digest d_i and extension proof π_i for each of the N intermediate logs ($i \in \{1, \ldots, N\}$). The service provider then sends the root R of a Merkle-tree commitment to these digests to each HSM.

Each HSM then asks the service provider for a random λ-size subset of the intermediate digests and extension proofs, where λ is a security parameter. The service provider returns the requested digests and extension proofs and proves that these values are included in the Merkle root R. Each HSM checks its requested intermediate extension proofs using DoesExtend$(\cdot)$ and checks the Merkle proof relative to the root R. The HSMs auditing the first and last chunks also ensure that the intermediate digests match the old digest d and the new digest d', respectively.

If these extension and Merkle proofs are valid, each HSM signs the tuple (d, d', R) using an aggregate signature scheme [14], and returns the signature to the service provider. Once all online HSMs have signed, the service provider aggre-

gates these signatures and broadcasts the aggregated signature to all HSMs. If any HSM fails during this process, the service provider notifies the HSMs and they restart this log-update process. (In the full version [24], we describe how the log can make progress even if HSMs fail during the log-update protocol.)

The HSMs check the aggregate signature on (d, d', R) relative to the HSMs' aggregate public key. If the signature is valid, the HSMs accept the new digest d'.

Security. If there are at most f_{secret} compromised HSMs, then even if f_{secret} honest HSMs are slow, $(1-2f_{\mathsf{secret}})N$ honest HSMs will participate in any successful protocol execution. If each of these HSM audits C chunks, then the probability that no honest HSM audits a particular log chunk is

$$\Pr[\text{fail}] = \left(1-\tfrac{1}{N}\right)^{(1-2f_{\mathsf{secret}})N\cdot C} \leq \exp\left((2f_{\mathsf{secret}}-1)\cdot C\right).$$

(Here, we use the fact that $(1-x) \leq \exp(-x)$.) If each HSM audits $C = \lambda \approx 128$ chunks, this failure probability is $\ll 2^{-128}$. In other words, some honest HSM will catch a cheating service provider with overwhelming probability. In addition, since all honest HSMs will expect a signature from all honest HSMs, this will cause the updating operation to fail and the system to halt. For this analysis, we assume that the adversary cannot adaptively compromise HSMs while the recovery protocol is running without taking them offline.

Scalability. Each HSM must check the extension proofs on λ chunks, where each chunk contains a $1/N$ fraction of the total updates in each epoch. Thus each HSM checks a vanishing fraction ($\frac{\lambda}{N}$) of log insertions. Each HSM checks one aggregate signature, which requires time independent of the number of HSMs [14]. Thus, the total work that each HSM performs per epoch decreases as the number of HSMs N increases.

Because we use the log primarily to limit the number of PIN attempts, garbage collection is straightforward. The service provider simply creates a new empty log, effectively resetting the number of PIN attempts for every user (old copies of the log can still be inspected to monitor recovery attempts). To ensure that the service provider does not run garbage collection and clear the state too frequently, each HSM will run garbage collection for a fixed number of times (e.g. the expected number of garbage collections over two years) before refusing to respond to further requests. This bounds the number of times the service provider can garbage collect the log.

6.3 Transparency and external auditability

Our log design allows *anyone* to audit the log to ensure that the service provider correctly maintains the log's append-only property. Additional auditors only add to the security of the system by adding another layer of protection, as they can detect log corruptions in the event that more than f_{secret} HSMs are compromised. In particular, for any two log digests d and d', an auditor can ask the data center for the entire logs L and L' corresponding to both of these digests. The auditor confirms that d is the root of the log tree for L and that d' is the root of the log tree for L'. Finally, the auditor checks that L' extends L.

As an extra precaution, users could specify external parties (e.g., Let's Encrypt) as designated auditors during backup. During recovery, the HSMs would only complete the recovery if these auditors sign the latest log digest. In this way, mounting a brute-force PIN-guessing attempt against a user would require compromising the user's external auditors as well.

The transparency log can also help with PIN re-use. As discussed in Section 8, instead of storing the salt directly with the service provider, the salt itself can be encrypted using a second round of location-hiding encryption and a null PIN. After recovery, the salt will be destroyed as discussed in the next section. Once the salt has been destroyed, the device restoring a backup can use the log to determine if anyone else has ever fetched the salt. If not, then it is safe for the user to re-use the old PIN.

As described in Section 4, the log contains usernames, which could be sensitive. To prevent leaking usernames, we would replace usernames with random device identifiers that are rerandomized when the device is factory reset. However, even with this modification, the log still leaks information about when and how often users restore backups, which the service provider may not wish to make public. While we hope that organizations would make their logs public, we acknowledge that some may only share their logs with several hand-picked organizations for auditing or may not share their logs at all. In these cases, our security guarantees still hold, although some of the transparency benefits are lost.

7 Forward security by puncturable encryption

We would like our encrypted-backup system to provide forward secrecy [18]. During the recovery process, the client reveals the identity of the $n \ll N$ HSMs that can decrypt its backup. Without forward secrecy, an attacker can break into these n HSMs to recover the client's backed-up data. Forward secrecy ensures that after recovery, an attacker, even one who compromises all HSMs in the data center, learns no information about the client's backup.

One seemingly straightforward way to provide forward secrecy would be to use a new keypair for each backup. However, because the client cannot interact with the HSMs it is encrypting to during backup (as this would reveal their identities), using a unique keypair for every backup would require every HSM in the data center to generate a new keypair for every backup, running counter to our scalability goals.

7.1 Background: Puncturable encryption

We instead achieve forward secrecy using puncturable public-key encryption [38, 39, 25, 21, 19, 26]. A puncturable encryption scheme is a normal public-key encryption scheme $(\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$, with one extra routine:

$\mathsf{Puncture}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{sk}_{\mathsf{ct}}$. Given a decryption key sk and a ciphertext ct, output a new secret key $\mathsf{sk}_{\mathsf{ct}}$ that can decrypt

all ciphertexts that sk could decrypt except for ct.

Puncturable encryption for forward secrecy. To achieve forward security in SafetyPin, after an HSM decrypts its share of a client's recovery ciphertext ct, the HSM *punctures* its secret decryption key. The punctured key allows the HSM to decrypt all ciphertexts except for ct. Thus, if an attacker compromises *all* HSMs in the data center after a client has recovered its backup, the attacker will be unable to decrypt any backup images that clients have already recovered. Furthermore, if an attacker compromises at most $f_{\mathsf{secret}} \cdot N$ HSMs total, where f_{secret} is a parameter of the system that we define in Section 3, then the attacker will not be able to recover any backed-up data whatsoever.

Existing tool: Bloom-filter encryption. Our implementation uses a puncturable encryption scheme called Bloom-filter encryption [25]. There are only two details of Bloom-filter encryption that are important for this discussion.

1. *The secret key is large.* If a key supports $P \in \mathbb{Z}^{>0}$ punctures and we want decryption to fail with probability at most $2^{-\lambda}$, then the secret key for Bloom-filter encryption is an array of roughly λP elements of a cryptographic group $\mathbb{G}$. After P punctures, the secret key may no longer decrypt messages and it is necessary to rotate encryption keys.
2. *Puncturing is simple.* Puncturing the secret key just requires deleting λ elements in the data array that comprises the secret key.

Concretely, when we set the Bloom-filter-encryption parameters to suitable values for experimental evaluation, each Bloom-filter encryption secret key has size over 64 MB. Even high-end HSMs have only 1–2 MB of storage (Table 2), so storing such large keys on an HSM would be impossible.

7.2 Outsourced storage with secure deletion

We show how to efficiently outsource the storage of this large secret key in a way that preserves forward secrecy of the punctured key. In particular, the HSM can outsource the storage of its secret-key array to the untrustworthy service provider, while still retaining the ability to delete portions of the key. Our technique applies to outsourcing the storage of any data array—not just secret keys—so we describe our secure-deletion approach in general terms.

Desired functionality. At a high level, the HSM has access to (a) a small amount of internal storage and (b) a large external block store, run by the service provider. The HSM wants to store an array of D data blocks at the provider $(\mathsf{data}_1, \ldots, \mathsf{data}_D)$. The HSM should be able to subsequently *read* or *delete* these blocks.

The following security properties should hold, even if the attacker, controlling the service provider, may choose the data-array and sequence of operations the HSM performs:

- **Integrity.** If the service provider tampers with the stored data in a way that could cause a *read* to return an incorrect result, the read operation outputs $\perp$. Otherwise, the read operation for a block i returns the value of the last data that the client wrote to block i.
- **Secure deletion.** If the service provider compromises the HSM after the HSM has run the *delete* operation for the ith data block, the attacker learns nothing about the data stored in block i. (This property implies a confidentiality property: the service provider learns nothing about the outsourced data.)

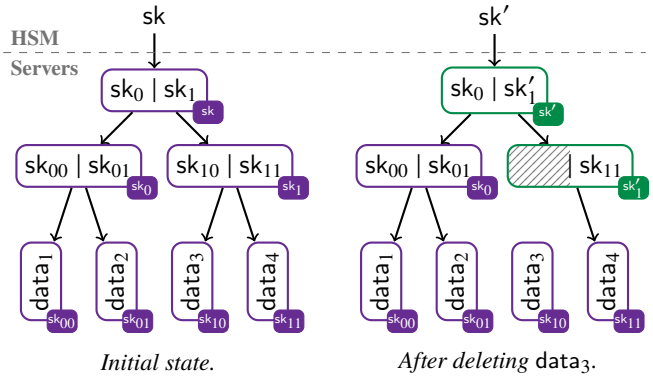


Figure 6: Our outsourced-storage scheme uses a tree of keys. An arrow $a \rightarrow b$ denotes that value b is stored encrypted under key a. A service provider that stores all values it sees and later compromises the HSM state (sk') still does not learn the deleted data_3 value.

For efficiency, the HSM storage requirements must be small (constant size) and the *read* and *delete* routines should run quickly (in time logarithmic in the size D of the data array). Unlike in ORAM [33, 34], our goal is not to hide the HSM's data-access pattern from the service provider. We aim only to hide the contents of the array.

7.3 Our secure outsourced storage scheme

We explain our construction here in prose. See the full version [24] for a more formal description.

Running the setup phase. During the setup phase, our outsourced-storage scheme builds a binary tree with D leaves. Every node of the tree contains a fresh symmetric encryption key. During setup, for each node in the tree with key sk_i, we encrypt the keys of the child nodes sk_{i0} and sk_{i1} with sk_i and store this ciphertext $\mathsf{AE.Encrypt}(\mathsf{sk}_i, \mathsf{sk}_{i0} \| \mathsf{sk}_{i1})$ in outsourced storage. At the leaves of the tree, we encrypt the ith data block with the key sk_i at the ith leaf and we store the ciphertext $\mathsf{AE.Encrypt}(\mathsf{sk}_i, \mathsf{data}_i)$ in outsourced storage.

For example, in Figure 6, we use sk_0 to encrypt sk_{00} and sk_{01} and we store the result in outsourced storage. We use key sk_{01} to decrypt data item 2. Thus, knowing the root key sk is enough to decrypt the entire tree and access every data element in the array.

Reading a data block. To retrieve the data block at index i, the HSM reads in the ciphertexts along the path from the tree root to leaf i. The HSM then decrypts the chain of ciphertexts

from the root down to recover the data block at index i. For example, in Figure 6, to retrieve data block 3, the HSM can use sk to decrypt sk_1, and sk_1 to decrypt sk_{10}, which it can use to decrypt data item 1.

Deleting a data block. To delete the data block at index i, the HSM recovers (as in retrieval) the keys along the path from the root to leaf i. At the node containing the key to decrypt data block i, the HSM deletes the key. It then chooses a fresh key and re-encrypts the other key at that node using the fresh key. To maintain the ability of the parent key to decrypt the child ciphertext, the HSM updates the parent of that node to contain the fresh key for its child and re-encrypts the parent's keys under a new key. It continues this up the path to the root, where the HSM chooses a new key sk' to encrypt the root. The HSM replaces sk with sk', deleting the old sk, and then sends the new ciphertexts along the path from the root to leaf i back to the service provider. For example, in Figure 6, to delete data item 3, the HSM decrypts the keys $(\mathsf{sk}_0 \| \mathsf{sk}_1)$ and $(\mathsf{sk}_{10} \| \mathsf{sk}_{11})$. The HSM then deletes sk_{10}, chooses a new key sk'_1 to encrypt sk_{11}, and then chooses a new key sk' to encrypt sk_0 and sk'_1. The HSM then replaces sk with sk'.

Efficiency. The setup time is linear in the size of the data array D. The runtimes of retrieval and deletion are both logarithmic in D, and require only symmetric-key operations. The HSM stores only the constant-sized root encryption key sk.

Security intuition. An HSM can always recover the keys necessary to decrypt a data item, provided the HSM did not previously delete any of the keys necessary for decryption. Integrity follows immediately from the security of the underlying authenticated encryption scheme. Finally, we ensure secure deletion by deleting the key necessary to decrypt a certain data item and updating the root key. Without the old root key, it is impossible to access the key necessary to decrypt the deleted data item.

Putting it together. To summarize: the HSMs use a puncturable encryption scheme to prevent the compromise of HSM secrets at time T from allowing an adversary to learn about backed-up data that was recovered any time before T. We implement puncturable encryption using Bloom-filter encryption and outsource the storage of the large secret decryption key using our new technique for outsourcing with secure deletion.

8 Extensions and deployment considerations

The full SafetyPin implementation has to deal with a number of additional issues, which we discuss now.

Failure during recovery. As discussed in Section 7, after participating in recovery, HSMs revoke their ability to decrypt the recovered ciphertext. One consequence is that a client cannot recover the same backup ciphertext twice. This raises the question of what happens if a replacement device fails during or shortly after recovery, or if a communication failure during recovery prevents the new device from receiving the replies from the HSMs.

To solve this problem, when a client initiates recovery, it first generates a fresh per-recovery keypair $(\mathsf{sk}, \mathsf{pk})$ for a public-key encryption scheme. The client backs up this secret key sk using SafetyPin before initiating its recovery. Next, the client sends the public key pk to each HSM and then begins the backup-recovery process. Each HSM encrypts its replies to the client under pk, and each HSM sends a copy of each reply to the data center. If a client device fails during recovery, a second, replacement client device can retrieve the backed-up secret key sk and use these to decrypt the replies stored at the data center. This scheme nests arbitrarily, thereby handling any number of consecutive device failures during recovery.

Incremental backups. In practice, mobile devices often generate incremental backups rather than encrypting the entire disk image for each backup. SafetyPin supports incremental backups in the following way. The user uses SafetyPin to store a single AES key, which the user also keeps on her phone. The user can then encrypt incremental backups under this AES key and upload the resulting ciphertext to the data center. When the user recovers, she recovers her AES key and can use this key to decrypt the incremental updates.

Multiple recovery ciphertexts. Clients back up their phones regularly (e.g., every three days), and will thus generate a series of recovery ciphertexts. We want to ensure that after a client recovers her backup from time t, the HSMs involved in recovery puncture their secret decryption keys so that they cannot decrypt that client's backups from earlier times $t' < t$, even if an attacker compromises all HSMs in the data center. To achieve this, in the puncturable-encryption step (Section 7), we have the client use the same salt for each recovery ciphertext it generates. In this way, the client will encrypt its series of backups to the same set of HSMs. When these HSM puncture their secret keys during the recovery process, they will destroy their ability to decrypt any previous recovery ciphertexts from the given client. After recovery, the client chooses a new salt to generate subsequent backups on its new device.

Preventing post-recovery PIN leakage. As we have discussed, an attacker that watches the client recover can learn a salted hash of the user's PIN, which can be used to mount an offline brute-force attack to learn the user's PIN.

One approach to protect against this attack would be to have each user store their salt in secret-shared form at a random set S_{salt} of HSMs, where S_{salt} is included in the client's recovery ciphertext. Then, provided that the attacker does not compromise this set of HSMs, the attacker would learn no useful information on the user's PIN, even after recovery. An attacker could always compromise every HSM in S_{salt}, but an attacker that can compromise only a f_{secret} fraction of HSMs in the data center would not be able to mount this attack against too many clients' salts. We hope to model and prove this multi-user PIN-protection property in future work.

Operation	Ops/sec
Pairing	0.43
ECDSA ver	5.85
ElGamal dec	6.67
$g^x \in \mathbb{G}_{P256}$	7.69

	Operation	Ops/sec
	HMAC-SHA256	2,173.91
	AES-128	3,703.70
I/O	RTT, HID (32b)	71.43
I/O	RTT, CDC (32b)	2,277.90
I/O	Flash read (32b)	≈166,000

Table 7: Microbenchmarks on SoloKey. Pairing is on BLS12-381 curve using the JEDI library [49]. Other public-key operations use NIST P256 curve.

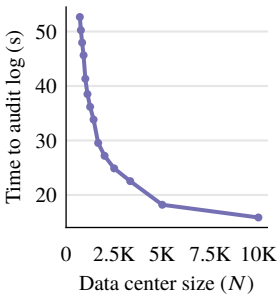


Figure 8: Log-audit time after inserting 10K recovery attempts for a log with roughly 100M recovery attempts. We only measure the auditing time for 100 HSMs as we only had 100 SoloKeys; we distribute the work as if there were N HSMs.

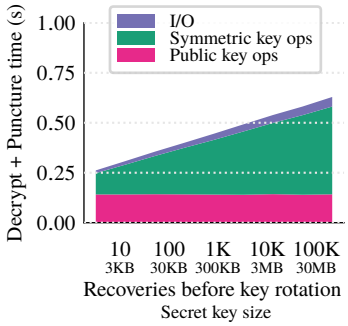


Figure 9: Time to run puncturable encryption on a single HSM as the maximum number of allowed punctures (and also secret key size) grows. The cost of our outsourced storage scheme dominates, though the access time is logarithmic in the size of the key.

9 Implementation and evaluation

We implemented SafetyPin on an experimental data cluster of 100 hardware security devices (Figure 1).

HSM. For the HSMs, we used SoloKeys [72], a low-cost open-source USB FIDO2 security key. SoloKeys use a STM32L432 microcontroller with an ARM Cortex-M4 32-bit RISC core clocked at 80MHz and 265KB of memory. The device is not side-channel resistant, but has a true random number generator and can lock its firmware. We add roughly 2,500 lines of C code to the open-source SoloKey firmware [71].

By default, SoloKeys communicate with the USB host via USB HID, an interrupt-based USB class used typically for keyboards and mice that has a maximum throughput of 64KBps. To improve performance, we rewrote parts of the firmware to use USB CDC, a high-throughput USB class commonly used for networking devices. This gave a roughly 32× increase in I/O throughput (Table 7).

For the puncturable-encryption scheme (Section 7.1), we use a variant of Bloom-filter encryption [25] that avoids the need for pairings [13] but increases the size of the HSMs' public keys. For the aggregate signature scheme needed for the log, we use BLS-style multisignatures [12] over the JEDI [49] implementation of the BLS12-381 curve.

Our implementation does not encrypt communication between the client and HSMs. Based on the time to run AES-128 and ElGamal encryption on the SoloKeys, we estimate that transport-layer encryption would add two ElGamal decryptions and 2KB of AES operations per recovery, increasing recovery time by approximately 0.3 seconds, or 30%. This overhead is comparatively high because processing a recovery only requires a handful of symmetric and public key operations.

Service Provider. Our service provider host is a Linux machine with an Intel Xeon E5-2650 CPU clocked at 2.60GHz. Our service-provider implementation is roughly 3,800 lines of C/C++ code (excluding tests) and uses OpenSSL.

Client. Our client device is a Google Pixel 4. Our implementation is roughly 2,300 lines of C/C++ code (excluding tests) and uses OpenSSL.

9.1 Microbenchmarks

Log. Figure 8 demonstrates how increasing the number of HSMs reduces the log-digest update time. We assume that the log is periodically garbage collected (i.e., approximately once a month), so that it holds at most a hundred million recovery attempts at once. If the HSMs run the log-update process every 10 minutes, each HSM spends approximately 11% of its active cycles auditing the log. The choice of how often to update the log is a tradeoff between how long users must wait to recover their backups and the total number of write cycles to non-volatile storage permitted by the hardware.

Puncturable encryption. Figure 9 shows the cost of performing a decrypt-and-puncture operation as the number of supported punctures increases. The AES operations associated with our scheme for outsourced storage with secure deletion (Section 7.2) dominate the cost.

Another way to implement outsourced storage with secure deletion would be to have the HSM store the outsourced array encrypted under a single AES key k. To delete an item, the HSM would read in the entire array, delete the item, and write out the entire array encrypted under a fresh key k'. With this approach, a deletion takes 48 minutes for a 64 MB array (the size of our outsourced secret keys). Our scheme thus improves system throughput by roughly $4,423\times$.

Each HSM punctures its secret key (Section 7.1) once after each decryption it performs. Since our puncturable-encryption scheme only supports a fixed number of punctures, each HSM must periodically rotate its encryption keys. We configure our puncturable-encryption scheme to allow each HSM to perform roughly 2^{18} decryptions before it must rotate its keys (rotation is triggered when half of the elements of the secret key have been deleted). Key rotation is expensive: we estimate (based on the number of public-key operations required) that key rotation on our HSMs will take roughly 75 hours. Each HSM spends approximately 139.4 hours processing recoveries and maintaining the log between key rotations. Therefore, each HSM spends roughly 56% of its cycles rotating its keys, and

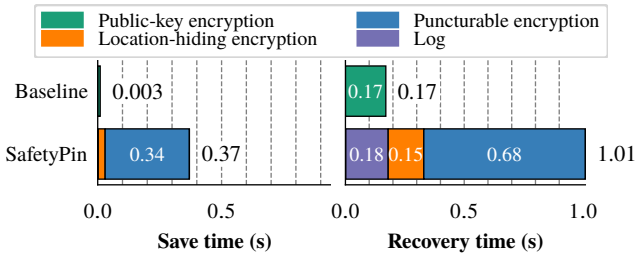


Figure 10: Breakdown of time to save (on Android Pixel 4 phone) and recover (using our SoloKey cluster). We do not consider the time to encrypt or decrypt disk images.

each HSM can process 1,503.9 recoveries per hour on average.

9.2 End-to-end costs

Parameters. We estimate that on average, each user will run recovery once a year. (There are 3.8B smartphone users [73] and 1.5B smartphones sold annually [74], so we expect $1.5/3.8 = 0.39 \ll 1$ recovery/user/year.) We calculate that a SafetyPin deployment of $N = 3,100$ HSMs could support one billion users. So, we treat our small cluster of 100 HSMs as a representative slice of a larger data center of $N = 3,100$ HSMs. Within this larger data center, each client shares its recovery keys among a cluster of $n = 40$ HSMs. This choice of n is based on the size of the data center N and PINs with six decimal digits, and is dictated by bounds we prove in the full version [24]. We set the puncturable encryption keys to allow 2^{20} punctures, as we found this provides a reasonable tradeoff between the time to decrypt and puncture and the time between key rotations. With these parameters, we maintain secrecy if at most an $f_{\mathsf{secret}} = \frac{1}{16}$ fraction of the HSMs are compromised (or $f_{\mathsf{secret}} \cdot N \approx 194$ total). We allow data recovery if at most an $f_{\mathsf{live}} = \frac{1}{64}$ fraction fail due to benign hardware failures (or $f_{\mathsf{live}} \cdot N \approx 48$ total).

Baseline. We compare against an encrypted-backup system modeled on the ones that Google and Apple use [82, 47]. To backup, the client selects a fixed cluster of five HSMs and encrypts her recovery key and a hash of her PIN under the cluster's public key. At recovery, the client sends the recovery ciphertext and a hash of her PIN to the cluster, and any HSM in the cluster can decrypt the ciphertext, check that the PIN hashes match, and return the recovery key. To defeat brute-force PIN-guessing attacks, each HSM independently limits the number of recovery attempts allowed on a given ciphertext.

Client overhead. Figure 10 gives the overhead of generating a backup in SafetyPin, compared to the baseline. The backup process takes 0.37 seconds. SafetyPin recovery ciphertexts are 16.5KB, versus 130B for our baseline, though we expect encrypted disk image to dominate the ciphertext size.

SafetyPin increases the bandwidth cost at the client. In the baseline scheme, the client downloads five public keys—one from each of its five chosen HSMs. In SafetyPin, the client must fetch a copy of all HSMs' public keys. (This way, the service provider does not learn the subset of HSMs to which the client is encrypting its backup.) So, when a client first joins the system, the client must download all these keys (11.5MB). Whenever an HSM rotates its puncturable-encryption keys, clients must download the HSM's new public key. In a deployment of $N = 3,100$ HSMs supporting one billion recoveries annually, we estimate that each SafetyPin client must download 1.97MB of keying material daily. Increasing the puncturable encryption failure probability would decrease client bandwidth, although this would require decreasing the fraction of HSMs allowed to fail, f_{live}. If a client goes offline for several days, it must download the rotated public keys for each day it spent offline (roughly 2MB/day), up to a maximum of 11.5MB (the size of all HSMs' keys). However, the client only needs to store the public keys for the n HSMs comprising its chosen recovery cluster which amounts to 9.02KB.

Recovery time. At a cluster size of $n = 40$ HSMs, Figure 11 shows that the end-to-end recovery time takes 1.01 seconds. Puncturable-encryption operations dominate recovery time (Figure 10), since these require expensive elliptic-curve operations for ElGamal decryption and many I/O and AES operations in order to perform secure deletion (Section 7.2).

Tail latency. In a deployment of SafetyPin, it will be important to consider not only the average throughput of the SafetyPin cluster, but also the request latency. Since recovery requests will arrive concurrently and in a bursty fashion, we will need to overprovision the system slightly to ensure that request tail latency does not grow too high, even under large transient loads. In Figure 13, we model how many HSMs are required to achieve various 99th-percentile latencies, while handling different average throughputs. We compute these values by modeling incoming requests using a Poisson process and each HSM using a M/M/1 queue with service times derived from our experimental results. As the figure demonstrates, by increasing the total number of HSMs, we can reduce the tail latency even when accounting for request contention. We anticipate that recovery time will in practice be dominated by the time to download the encrypted disk image, and so as long as the tail latency is less than or close to this time, any delay is unlikely to be noticed by the user.

Financial cost. Figure 12 shows how throughput scales as the outlay on HSMs increases and Table 14 presents dollar-cost estimates for SafetyPin deployments with different types of HSMs. For a configuration that tolerates the compromise of 50 high-quality HSMs, we estimate that adding SafetyPin to an unencrypted backup system would increase the system's dollar cost by 2.5%.

10 Related work

Today's encrypted-backup systems rely either on the security of hardware security modules [37, 47], secure microcontrollers [3], or secure enclaves [55, 57]. Vulnerabilities in these hardware components leave encrypted-backup systems

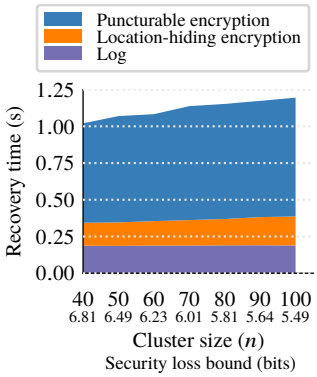


Figure 11: Recovery time grows slowly as cluster size n increases. The bits of security lost refers to the difference between the advantage of an attacker against a SafetyPin deployment with the given value of n and an attacker trying to guess the user's PIN.

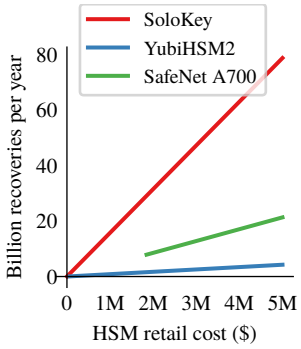


Figure 12: Estimated number of SafetyPin-protected recoveries per year supported by clusters of different HSM models and costs. We use g^x/sec to compute the expected throughput of more powerful HSMs based on our measurements using SoloKeys (Table 2).

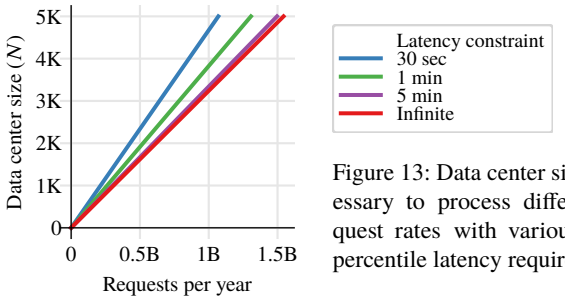


Figure 13: Data center sizes necessary to process different request rates with various 99th-percentile latency requirements.

open to attack. And there is ample evidence of vulnerabilities in both HSMs [66, 60, 63, 41, 46, 17, 31, 2] and enclaves [79, 16, 36, 53, 28, 80, 81, 61, 20, 40, 52, 11], and reason for concern about hardware backdoors as well [76, 83, 7, 48].

Many companies including Anchorage [5], Unbound Tech [78], Curv [22], and Ledger Vault [51], offer systems for secret-sharing cryptocurrency secret keys across multiple hardware devices. Unlike SafetyPin, these solutions use a small fixed set of HSMs, so they cannot simultaneously provide scalability and protection against adaptive HSM compromise.

In recent theoretical work, Benhamouda et al. show how to scalably store secrets on proof-of-stake blockchains when an adversary can adaptively corrupt some fraction of the stake [10]. They face many of the same cryptographic challenges that we tackle in Section 5; their theoretical treatment complements our implementation-focused approach. While they use proactive secret sharing to periodically re-share the secret and hide the secret from an adversary controlling some fraction of the stake, our approach allows a party with some low-entropy secret to recover the high-entropy secret.

Transparency logs inspire our log design [6, 50, 58, 1, 54]. While these logs allow a powerful auditor to verify correctness,

	HSM Qty.	f_{secret}	N_{evil}	Cost
SoloKey [72]	3,037	1/16	189	$60.7K
YubiHSM2 [84]	1,732	1/16	108	$1.1M
SafeNet A700 [68]	40	1/20	2	$738.7K
– 10 evil HSMs	320	1/32	10	$3.0M
– 50 evil HSMs	800	1/16	50	$14.8M
Estimated cost of storing 4GB $\times 10^9$ *users per year:*				$600M

Table 14: The estimated hardware cost of a SafetyPin deployment supporting one billion users, if each user recovers once per year. The N_{evil} number is how many corrupt HSMs the deployment tolerates.

We estimate the storage cost using AWS S3 infrequent access [4] ($0.0125 per GB/month). We estimate YubiHSM2 and SafeNet HSM throughput using their data sheets (Table 2). When computing the number of HSMs necessary to service a billion users, we account for key-rotation time. A cluster of 40 SafeNet HSMs can meet the throughput demands of one billion users, so we also consider larger deployments tolerating more compromised HSMs.

they do not easily allow distributing the work of auditing across many less powerful participants. The proofs we provide to the HSMs about the state of the log draw on work on authenticated data structures [75, 56, 65] and cryptocurrency light clients [62]. Kaptchuk et al. show how public ledgers can be used to build stateful systems from stateless secure hardware [45], and they show how their techniques can be applied to Apple's encrypted-backup system. This work is complementary to ours, as they show how to securely manage state in cases where HSMs do not have secure internal non-volatile storage (an assumption we make in SafetyPin).

11 Conclusion

SafetyPin is an encrypted backup system that (a) requires its users to only remember a short PIN, (b) defeats brute-force PIN-guessing attacks using hardware protections, and (c) provides strong protection against hardware compromise. SafetyPin demonstrates that it is possible to reap the benefits of hardware security protections without turning these hardware devices into single points of security failure.

Acknowledgments. We would like to thank Raluca Ada Popa and Bryan Ford for their support throughout this project. Albert Kwon, Anish Athalye, Christian Mouchet, David Lazar, Dima Kogan, and Lefteris Kokoris-Kogias, offered thoughtful criticism on drafts of this work. We thank our shepherd Sebastian Angel for his work reviewing our camera-ready. We thank Dan Boneh for useful suggestions on how to simplify the security analysis of our location-hiding encryption scheme, we thank Vinod Vaikuntanathan for the suggestion to avoid pairings in our puncturable-encryption scheme, and we thank Keith Winstein for early brainstorming on passwords and PINs. Conor Patrick and Nicolas Stalder gave helpful suggestions for modifying the SoloKey firmware to support USB CDC, and Vivian Fang provided advice on assembling the system. Finally, we thank the anonymous reviewers of USENIX Security 2020 and OSDI 2020 for their thorough and detailed feedback. This research is also supported in part by the RISELab, a Facebook Research Award, and a NSF GRFP fellowship.

References

[1] Trillian. https://github.com/google/trillian.

[2] CVE-2015-5464. Available from MITRE, CVE-ID CVE-2015-5464., February 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5464.

[3] Titan in depth: Security in plaintext. Google, August 2017. https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext.

[4] Amazon S3 pricing. https://aws.amazon.com/s3/pricing/, Accessed 12 February 2020.

[5] Anchorage. https://anchorage.com/, Accessed 25 May 2020.

[6] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E Culler, and Raluca Ada Popa. WAVE: A decentralized authorization framework with transitive delegation. In *USENIX Security*, 2019.

[7] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In *CHES*. Springer, 2013.

[8] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2001.

[9] Mihir Bellare, Dennis Hofheinz, and Scott Yilek. Possibility and impossibility results for encryption and commitment secure under selective opening. In *EUROCRYPT*, pages 1–35. Springer, 2009.

[10] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a blockchain keep a secret? *IACR Cryptology ePrint Archive*, 2020.

[11] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *USENIX Security*, pages 1213–1227, 2018.

[12] Dan Boneh, Manu Drijvers, and Gregory Neven. BLS multi-signatures with public-key aggregation. https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html, Accessed 23 May 2020, 03 2018.

[13] Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, pages 213–229, 2001.

[14] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, pages 416–432. Springer, 2003.

[15] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. 2020.

[16] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX WOOT*, 2017.

[17] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. Bios chronomancy: Fixing the core root of trust for measurement. In *CCS*, pages 25–36. ACM, 2013.

[18] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT*, pages 255–271, 2003.

[19] Ran Canetti, Srinivasan Raghuraman, Silas Richelson, and Vinod Vaikuntanathan. Chosen-ciphertext secure fully homomorphic encryption. In *IACR International Workshop on Public Key Cryptography*, pages 213–240. Springer, 2017.

[20] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE: Stealing intel secrets from SGX enclaves via speculative execution. In *EuroS&P*, pages 142–157. IEEE, 2019.

[21] Aloni Cohen, Justin Holmgren, Ryo Nishimaki, Vinod Vaikuntanathan, and Daniel Wichs. Watermarking cryptographic capabilities. *SIAM*, 47(6):2157–2202, 2018.

[22] Curv. https://www.curv.co/, Accessed 25 May 2020.

[23] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, 2014.

[24] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. SafetyPin: Encrypted backups with human-memorable secrets. *arXiv preprint arXiv:2010.06712*, 2019.

[25] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In *EUROCRYPT*, pages 425–455. Springer, 2018.

[26] David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. Revisiting proxy re-encryption: forward secrecy, improved security, and applications. In *IACR International Workshop on Public Key Cryptography*, pages 219–250. Springer, 2018.

[27] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[28] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[29] Serge Fehr, Dennis Hofheinz, Eike Kiltz, and Hoeteck Wee. Encryption schemes secure against chosen-ciphertext selective opening attacks. In *EUROCRYPT*, pages 381–402. Springer, 2010.

[30] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *Symposium on Operating Systems Design and Implementation*, pages 233–248, 2006.

[31] Jean-Baptiste Bédrune Gabriel Campana. Everybody be Cool, This is a Robbery! Blackhat, 2019. https://www.blackhat.com/us-19/briefings/schedule/#everybody-be-cool-this-is-a-robbery-16233.

[32] Shirley Gaw and Edward W Felten. Password management strategies for online accounts. In *SOUPS*, pages 44–55. ACM, 2006.

[33] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.

[34] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[35] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[36] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *European Workshop on Systems Security*, pages 1–6, 2017.

[37] Matthew Green. Is Apple's Cloud Key Vault a crypto backdoor?, 2016. https://blog.cryptographyengineering.com/2016/08/13/is-apples-cloud-key-vault-crypto/.

[38] Matthew D Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *Security and Privacy*. IEEE, 2015.

[39] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 2017.

[40] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, pages 44–60. Springer, 2018.

[41] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. A bad dream: Subverting trusted platform module while you are sleeping. In *USENIX Security*, pages 1229–1246, 2018.

[42] SM Haque, Matthew Wright, and Shannon Scielzo. A study of user password strategy for multiple accounts. In *Data and application security and privacy*, pages 173–176. ACM, 2013.

[43] Dennis Hofheinz and Andy Rupp. Standard versus selective opening security: separation and equivalence results. In *Theory of Cryptography Conference*, pages 591–615, 2014.

[44] Intel. Strengthen enclave trust with attestation. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/attestation-services.html, Accessed 5 February 2020.

[45] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *NDSS*, 2019.

[46] Bernhard Kauer. Oslo: Improving the security of trusted computing. In *USENIX Security*, volume 24, page 173, 2007.

[47] Ivan Krstic. Behind the scenes with iOS security, 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf.

[48] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian. Parametric trojans for fault-injection attacks on cryptographic hardware. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sept. 2014.

[49] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E Culler. JEDI: Many-to-many end-to-end encryption and key delegation for IoT. In *USENIX Security*, pages 1519–1536, 2019.

[50] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency. *Internet Engineering Task Force*, 2013. https://tools.ietf.org/html/rfc6962.

[51] Ledger vault. https://www.ledger.com/vault, Accessed 25 May 2020.

[52] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[53] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[54] S. Li, C. Man, and J. Watson. Delegated Distributed Mappings. Internet-Draft draft-watson-dinrg-delmap-02, Internet Engineering Task Force, April 2019. https://tools.ietf.org/html/draft-watson-dinrg-delmap-02.

[55] Joshua Lund. Technology preview for secure value recovery, 2019. https://signal.org/blog/secure-value-recovery/.

[56] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[57] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 10(1), 2013.

[58] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

[59] Ralph Merkle. A certified digital signature. In *CRYPTO*, pages 218–238. Springer, 1989.

[60] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *USENIX Security*, 2020.

[61] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Security and Privacy*. IEEE, 2020.

[62] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.

[63] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of Coppersmith's attack: Practical factorization of widely used RSA moduli. In *CCS*, pages 1631–1648. ACM, 2017.

[64] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In *USENIX Security Symposium*, 1998.

[65] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *International conference on information and communications security*, pages 1–15. Springer, 2007.

[66] Ryan Paul. Infineon DRM/encryption chip succumbs to physical attack. Ars Technica, 2010. https://arstechnica.com/information-technology/2010/02/infineon-drmencryption-chip-succumbs-to-physical-attack/.

[67] Adam Powers. FIDO TechNotes: The truth about attestation. https://fidoalliance.org/fido-technotes-the-truth-about-attestation/, Accessed 25 May 2020.

[68] SafeNet Luna network hardware security modules A700 - cryptographic accelerator. https://www.insight.com/en_US/shop/product/908-000366-001-000/GEMALTO/908-000366-001-000/SafeNetLunaNetworkHardwareSecurityModulesA700-Cryptographicaccelerator-GigE-1U-rack-mountable/, Accessed 5 February 2020.

[69] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[70] Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Encountering stronger password requirements: user attitudes and behaviors. In *SOUPS*, page 2. ACM, 2010.

[71] Solokeys. Solo. https://github.com/solokeys/solo, Accessed 5 February 2020.

[72] Solokeys. https://solokeys.com/, Accessed 5 February 2020.

[73] Statista. Number of smartphone users worldwide from 2016 to 2021. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[74] Statista. Number of smartphones sold to end users worldwide from 2007 to 2020. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/, Accessed 23 May 2020.

[75] Roberto Tamassia. Authenticated data structures. In *European symposium on algorithms*, pages 2–5. Springer, 2003.

[76] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design Test of Computers*, 27(1), Jan 2010.

[77] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *CCS*, pages 1299–1316, 2019.

[78] Unbound tech. https://www.unboundtech.com/, Accessed 25 May 2020.

[79] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[80] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[81] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017.

[82] Shabsi Walfish. Google Cloud Key Vault Service. Google, 2018. https://developer.android.com/about/versions/pie/security/ckv-whitepaper.

[83] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. A2: Analog malicious hardware. In *Security and Privacy*. IEEE, May 2016.

[84] YubiHSM 2. https://www.yubico.com/product/yubihsm-2, Accessed 5 February 2020.

A Artifact Appendix

A.1 Abstract

The SafetyPin implementation is split into two components:

- **HSM:** The HSMs (hardware security modules) are used to recover user secrets. Our implementation uses SoloKeys, which are low-cost HSMs. We add roughly 2,500 lines of C code to the open-source SoloKey firmware.
- **Host:** The host implements functionality for the user and data center, including saving secrets, maintaining the log, and coordinating HSMs. Our implementation is roughly 3,800 lines of C/C++ code.

We implement the protocol described in the paper above. To improve performance, we rewrote parts of the SoloKey firmware to use USB CDC, a high-throughput USB class commonly used for networking devices. This results in roughly a 32× increase in I/O throughput. Our artifact is available at:

`https://github.com/edauterman/SafetyPin`

A.2 Artifact check-list

- **Hardware:**
  - 100 SoloKeys
  - 10 Anker SuperSpeed USB 3.0 hubs
  - 2 4-port USB PCIe controller cards
  - Linux machine with Intel Xeon E5-260 CPU clocked at 2.60GHz
- **Compilation:** The ARM compiler for the SoloKeys, and `gcc` for the host.
- **Metrics:** Latency
- **Experiments:** Log-audit time, puncturable encryption overhead, breakdown of recovery time, cluster size vs. recovery time
- **Required disk space:** 14MB
- **Expected experiment run time:** 50 minutes
- **Public link:** `https://github.com/edauterman/SafetyPin`
- **Code licenses:** Apache v2

A.3 Description

A.3.1 How to access

We provide reviewers with credentials to remotely access our system. Instructions for assembling a similar system are available here:

`https://github.com/edauterman/SafetyPin/blob/master/SETUP.md`.

A.3.2 Hardware dependencies

Our artifact uses SoloKeys as low-cost HSMs. We use 100 SoloKeys for our experiments, although other deployments

could use a different number of HSMs. Because of limitations in the Intel XCHI controller, which supports a maximum of 96 endpoints (each USB 3.0 device has 3 endpoints), we installed PCIe cards to support additional endpoints. This is not necessary for smaller-scale deployments, but larger deployments should choose a host that supports installing such PCIe cards or find another solution. We also recommend USB hubs with an external power source such as the Anker hubs.

A.3.3 Software dependencies

The firmware for the SoloKeys builds on the original SoloKey firmware, which already includes several libraries for cryptographic primitives on embedded devices:

`https://github.com/solokeys/solo`.

To support pairings for aggregate signatures, we use the jedi-pairing library for embedded devices:

`https://github.com/ucbrise/jedi-pairing/`.

For USB HID support, we use the Signal11 library:

`https://github.com/signal11/hidapi`.

We implement our cryptographic primitives that do not require pairings at the host using OpenSSL.

A.4 Installation

Instructions for building the host are available under `host/`. Instructions for building the firmware and flashing the SoloKeys are available under `hsm/`. The SoloKey documentation provides additional details and troubleshooting for building and flashing the SoloKeys:

`https://docs.solokeys.io/`.

When experimenting with SafetyPin, you should not boot SoloKeys in DFU mode, as this locks the firmware and will prevent you from modifying the firmware later (e.g. to load an updated version of the SafetyPin source).

A.5 Experiment workflow

Reviewers can remotely access our machine and run all experiments by executing `./runAll.sh` in `bench/`. More detailed instructions for running individual experiments are available here:

`https://github.com/edauterman/SafetyPin#instructions-for-artifact-evaluation`.

A.6 Evaluation and expected result

Run all experiments by executing `./runAll.sh` in `bench/`. This will produce figures in `bench/out` that match Figure 8, Figure 9, Figure 10, and Figure 11. Note that we only reproduce the recovery time breakdown in Figure 10. Additionally, the configuration we set up for the reviewers only uses 90 HSMs for Figure 8 and Figure 11. We do this to keep different firmware on the remaining 10 HSMs to measure the breakdown in puncturable encryption time as the secret key size increases (Figure 9). For the experiments we show in the body of the paper, we re-flashed HSMs between experiments so that we could use all 100 HSMs to generate Figure 8 and Figure 11.

A.7 Experiment customization

The experiment for Figure 8 can be modified to measure different data center sizes without changing the firmware on the HSMs. The experiment for Figure 9 can likewise be modified to measure different secret key sizes, although this requires changing HSM firmware. If we had more HSMs, we could easily expand Figure 11 to show the effect of larger cluster sizes. We do not measure cluster sizes less than 40 because our analysis shows that our security guarantees begin to break down below this point.

A.8 Notes

To switch between USB CDC and USB HID, change the HID flag on both the host and the HSMs (this requires loading new firmware on the HSMs). More detailed instructions are available here:

`https://github.com/edauterman/SafetyPin/blob/master/SETUP.md`.

Note that rather than generating puncturable encryption secret keys on the HSM (a process we estimate would take roughly 75 hours), to run our experiments efficiently, we generate the secret key on the host (for security, a real-world deployment would need to generate this secret key on the HSM).

A.9 AE Methodology

Submission, reviewing and badging methodology:

`https://www.usenix.org/conference/osdi20/call-for-artifacts`



Efficiently Mitigating Transient Execution Attacks using the Unmapped Speculation Contract

Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract

Today's kernels pay a performance penalty for mitigations—such as KPTI, retpoline, return stack stuffing, speculation barriers—to protect against transient execution side-channel attacks such as Meltdown [21] and Spectre [16].

To address this performance penalty, this paper articulates the *unmapped speculation contract*, an observation that memory that isn't mapped in a page table cannot be leaked through transient execution. To demonstrate the value of this contract, the paper presents WARD, a new kernel design that maintains a separate kernel page table for every process. This page table contains mappings for kernel memory that is safe to expose to that process. Because a process doesn't map data of other processes, this design allows for many system calls to execute without any mitigation overhead. When a process needs access to sensitive data, WARD switches to a kernel page table that provides access to all of memory and executes with all mitigations.

An evaluation of the WARD design implemented in the sv6 research kernel [8] shows that LEBench [24] can execute many system calls without mitigations. For some hardware generations, this results in performance improvement ranging from a few percent (`huge page fault`) to several factors (`getpid`), compared to a standard design with mitigations.

1 Introduction

Over the last two years, transient execution has emerged as a powerful new side-channel attack technique. Vulnerabilities have proliferated [5, 12], with examples now including Meltdown [21], Spectre [16], L1 Terminal Fault [4], RIDL [29], Fallout [6], ZombieLoad [25], CrossTalk [23], and SGAxe [28]. In contrast with conventional timing-based side-channel attacks [17], where the victim must access its data in a specific pattern in order to leak it, transient execution attacks are more serious because they often allow an attacker to precisely control which memory locations are leaked, including memory that might not be accessed on the committed execution path. This is of particular concern to OS kernels, which have access to all of physical memory, and therefore could leak data from any process through transient execution bugs. In a public cloud, where it is common for mutually distrustful tenants to share a single machine [30, 35], the threat of transient execution is especially concerning.

A key challenge in addressing transient execution attacks lies in minimizing the performance overheads. CPU and OS designers have implemented a range of mitigations to defeat transient execution attacks, including state flushing, selectively preventing speculative execution, and removing observation channels [5]. These mitigations impose performance overheads (see §2): some of the mitigations must be applied at each privilege mode transition (e.g., system call entry and exit), and some must be applied to all running code (e.g., retpolines for all indirect jumps). In some cases, they are so expensive that OS vendors have decided to leave them disabled by default [2, 22]. Recent processor designs have also incorporated mitigations into hardware, which also reduces performance compared to earlier processor designs that do not perform such hardware mitigations.

To address the above challenge, this paper proposes a new hardware/software contract, called the *unmapped speculation contract*, or USC for short. USC allows the OS kernel to significantly reduce the overhead of mitigating a particular subset of transient execution attacks—namely, those that leak arbitrary memory contents. The USC says that physical memory that is unmapped (i.e., physical memory that has no virtual address) cannot be accessed speculatively. The benefit of USC is two-fold. From the OS designer perspective, it provides bounds on what data can be leaked through transient execution, and, as we show in the rest of this paper, can significantly reduce the cost of mitigations. From the hardware designer perspective, USC allows the CPU to keep many of the current speculative execution optimizations and their associated performance benefits. Most processor architectures already adhere to USC; AMD states that "AMD processors are designed to not speculate into memory that is not valid in the current virtual address memory range defined by the software defined page tables" [1, pg. 2], and Intel issued hardware and microcode fixes for bugs that violate USC [14, 15].

To demonstrate the benefits of the unmapped speculation contract, this paper presents WARD, a novel kernel architecture that uses selective kernel memory mapping to avoid the costs of transient execution mitigations. WARD maintains separate kernel memory mappings for each process, and ensures that the memory mapped in the kernel of a process does not contain any data that must be kept secret from that process. As a result, privilege mode switches (e.g., system call entry and exit) no longer need to employ expensive mitigations, since there are no secrets that could be leaked by transient execution. When the WARD kernel must perform operations that require access to unmapped parts of kernel memory, such as opening a shared file or context-switching between processes, it explic-

itly changes kernel memory mappings, and invokes the same mitigation techniques used by the Linux kernel today.

A key challenge in the WARD design lies in re-architecting the kernel and its data structures to allow for per-process views of the kernel address space. For example, a typical `proc` structure in the kernel contains sensitive fields, such as the saved registers of that process, which should not be leaked to other processes. At the same time, every process must be able to invoke the scheduler, which in turn may need to traverse the list of `proc` structures on the run queue. This paper presents several techniques to partition the kernel: transparent switching of kernel address spaces when accessing sensitive pages through page faults; using temporary mappings to access unmapped physical pages; splitting data structures into public and private parts; etc.

To evaluate the WARD design, we applied it to the sv6 research kernel [8] running on x86 processors. The sv6 kernel is a monolithic OS kernel written in C/C++, providing a POSIX interface similar to (but far less sophisticated than) Linux. The simplicity of sv6 allowed us to quickly experiment with and iterate on WARD's design, since some aspects of WARD's design require global changes to the entire kernel. Since sv6 is a monolithic kernel, our prototype was able to tackle hard problems brought up by kernel services such as a file system and a POSIX virtual memory system.

We evaluate the performance of our WARD prototype using LEBench [24], which represents the most important system calls for a range of application workloads: Spark, Redis, PostgreSQL, Chromium, and building the Linux kernel. LEBench allows us to precisely measure the impact of mitigations on system calls that matter for applications. The most recent Intel CPUs (such as Cascade Lake) include hardware mitigations that cannot be fully disabled; however, some of these mitigations are not needed in WARD. To avoid the performance overhead of such unnecessary mitigations, we run experiments on the previous generation of Intel CPUs (Skylake).

WARD can run the LEBench microbenchmarks with small performance overheads compared to a kernel without mitigations. For 18 out of the 30 LEBench microbenchmarks, WARD's performance is within 5% of the benchmark's performance without any mitigations (but at the cost of some extra memory overhead). In the worst case, the overhead is 4.3× (context switching between processes, where mitigations are unavoidable). In contrast, standard mitigations incur a median overhead of 19%, and a worst case of nearly 7×. To confirm that LEBench results translate into application performance improvements, we measured the performance of `git status`, which incurs 11.2% overhead in WARD, compared to 24.6% with standard mitigations.

One of the limitations of USC is that it does not cover all possible transient execution attacks. In particular, attacks where the sensitive information is already present in the architectural or microarchitectural state of the CPU are not covered by USC. For instance, the Spectre v3a attack can leak the sensitive contents of a system register (MSR), instead of leaking sensitive data from memory. USC does not cover sensitive data that is stored outside of memory, and WARD applies other mitigations (e.g., as in Linux) to address those attacks.

2 Motivation

Transient execution mitigations harm kernel performance in two ways. First, they place overhead on code execution by disabling speculation. For example, the Linux Kernel uses a retpoline patch to mitigate Spectre V2, which replaces each indirect branch with a sequence of instructions that prevent the CPU from performing branch target speculation [13]. Second, these mitigations increase the privilege mode switching cost incurred during each system call: upon entry into the kernel, they either flush microarchitectural state or reconfigure protection mechanisms. For example, KPTI [11, 20] switches to a separate page table before executing kernel code to prevent Meltdown attacks [21]. Workloads that are system call intensive (e.g., web servers, version control systems, etc.) are impacted by this type of overhead, while non-kernel intensive workloads see little performance impact [11].

Collectively, these and other mitigations can result in large slowdowns. To better understand this problem, we run LEBench [24], a microbenchmark suite of system calls that impact application performance the most. We evaluate the Linux kernel (version 5.6.13), comparing two configurations: one where all mitigations are disabled and one where all are enabled. Figure 1 shows the relative slowdown between the two configurations for 13 kernel operations of LEBench that don't involve networking (i.e., without `send`, `recv`, `epoll`). There are two sets of bars, representing two generations of Intel CPUs: the older Skylake, and the newer Cascade Lake. On the older Skylake CPUs, system calls that perform the least kernel work are impacted the most (e.g., `getpid()`), but a wide range of operations are impacted significantly (25%-100% slowdowns). These observations are similar to those made by Ren et. al.; they find that KPTI and Spectre V2 mitigations are the root cause of slowdowns in the Linux Kernel over the last two years [24].

The newer Cascade Lake CPUs exhibit lower relative overheads, partly because the processors include hardware mitigations for some of the transient execution vulnerabilities. However, these lower overheads are also in part due to the newer Cascade Lake CPUs being *slower* in the baseline case when software-controllable mitigations are disabled. Figure 2 shows the performance of the microbenchmark on Cascade Lake (Intel Xeon Silver 4210R) relative to the earlier Skylake CPU (Intel Xeon E5-2640 v4). Our experiment uses CPUs with identical clocks (2.4 GHz), and nearly identical other hardware (Dell PowerEdge T430 vs. T440), which allows the comparison to be meaningful. The results demonstrate that, although the new CPU is faster at some microbenchmarks, it is slower for many others: e.g., context-switching is about 20% slower. Although it is impossible for us to separate slowdowns

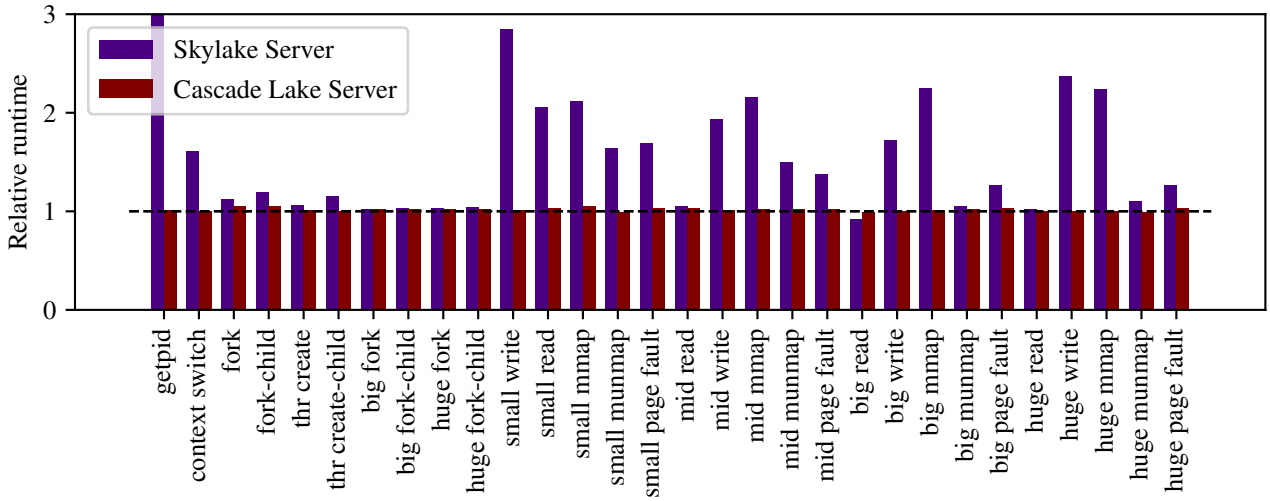


Figure 1: Linux slowdown due to mitigations on LEBench, for two generations of Intel CPUs: Skylake and Cascade Lake.

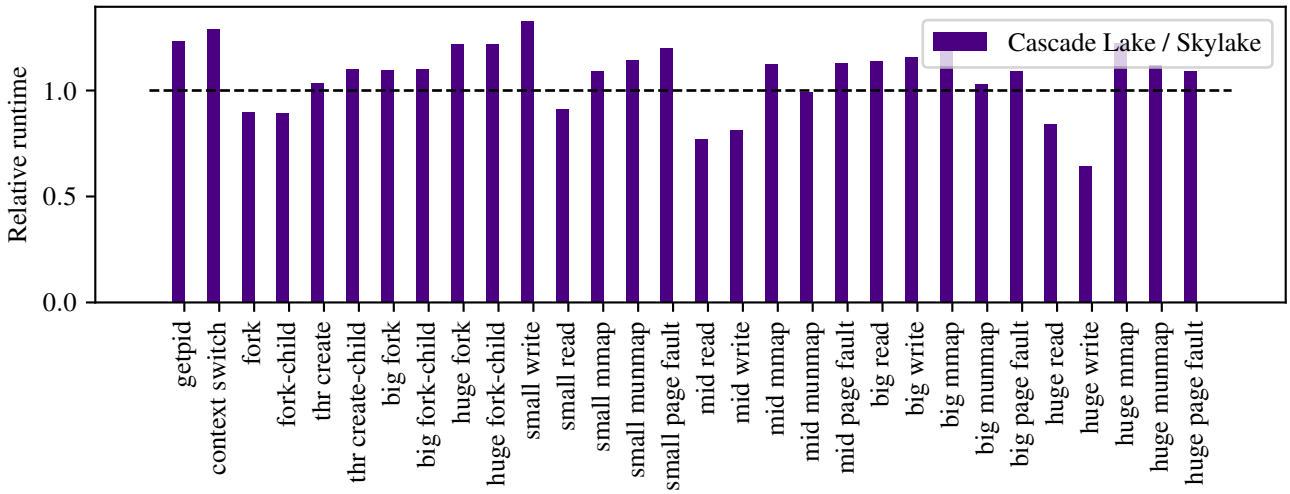


Figure 2: Performance regression on the newer Cascade Lake CPU, compared to the older Skylake CPU, for LEBench on Linux, with all software-controllable mitigations disabled.

due to mitigations from speedups due to architectural improvements, the results suggest that the overheads of mitigations implemented in hardware (e.g., for Meltdown, L1TF, or MDS) could still be significant. [1]

3 Goal and threat model

WARD's goal is to reduce the performance cost of mitigations for transient execution attacks. In principle, WARD's techniques can reduce not only the cost of software mitigations, but also allow processor designers to avoid costly mitigations in hardware. Practically, however, it is difficult for us to disable hardware mitigations in the newest processors. Therefore, this paper focuses on reducing the overhead of software mitigations, and experimentally measures their effect on the previous generation of CPUs, where we can avoid mitigations altogether. We hope that WARD's design can allow processor designers to regain some of the absolute performance lost due to hardware mitigation costs.

Our threat model targets scenarios where the adversary and the victim are both running code on the same computer. This might arise either in a server setting, where both are running on a cloud computing platform, or in a client device, where the adversary code is a malicious application or web site.

Canella et al. [5] discuss transient execution attacks in detail, but the salient points of the attack boil down to four steps. First, the processor speculatively executes some code, which accesses sensitive data that the victim wants to keep secret. Second, during the speculative execution, the processor updates microarchitectural state in a way that depends on the sensitive data (e.g., bringing in cache lines into a shared L3

[1] One indication that this regression may be related to hardware mitigations is that measured branch mispredictions are around 40% higher on LEBench.

cache whose addresses depend on the sensitive data). Third, the processor aborts the speculative execution, but does not fully roll back all of its side effects (e.g., changes to the L3 cache), because doing so would be prohibitively expensive in hardware. Fourth, the adversary observes these side effects (e.g., using timing measurements), which allows the adversary to infer the sensitive data.

What makes transient execution attacks challenging to mitigate in an OS kernel is a combination of two factors. The first is that an adversary can trigger an OS kernel to speculatively execute code that leads to leakage of sensitive data. Even though the adversary cannot inject their own code to execute in the kernel, the adversary can often have significant influence on what existing kernel code gets executed in speculative execution, by specifying particular system call arguments or setting up micro-architectural CPU state such as the branch predictor. The second factor is that an OS kernel has access to all of the state on the computer. This means that an adversary running in one process can trick the kernel into leaking state from any other process on the same computer.

Current OS kernel designs, such as Linux, have two approaches for mitigating transient execution attacks. The first approach is to make sure that the CPU does not speculatively execute any code that could end up accessing sensitive data. This approach includes techniques such as retpolines and other speculation barriers. The second approach is to make sure that sensitive data is flushed from microarchitectural state, such as flushing CPU caches and buffers when returning from a system call or when context-switching between processes. Both incur significant performance overheads.

Transient execution attacks can leak data across many protection domain boundaries, such as leaking secrets from the kernel to an adversary's process, or leaking secrets from one process to a different process, or even leaking secrets within a single process that implements its own internal protection domains. Much like in the Linux kernel, the focus of WARD is on preventing leakage between processes, as well as preventing leakage from the kernel to a process. WARD's approach to preventing cross-process leakage is the same as Linux (flushing state), but WARD has a novel approach for efficiently preventing kernel-to-process leakage of memory contents, as we describe in the next section.

Although WARD addresses all known transient execution attacks, the focus of this paper is on attacks that allow the adversary to leak the contents of arbitrary memory, which is especially important in an OS kernel. WARD handles other transient execution attacks, such as leaking the contents of sensitive data already present in the CPU (e.g., x86 MSRs), in the same way as Linux does.

Attacks that do not leverage transient execution to leak data are also out of scope for this paper, since they are orthogonal to the key challenge of transient execution leakage. In particular, we do not consider attacks that leverage physical side channels (such as Rowhammer or RAMbleed), cache side channels (such as cache timing attacks), power side channels, etc.

4 Approach: Unmapped speculation contract

WARD's design for mitigating transient execution attacks relies on page tables. Specifically, if a page of physical memory is not referenced by any entry in the current page table or TLB, speculative execution cannot access any sensitive data stored in that page, because the page doesn't have a virtual address to access it by.

A contribution of this paper lies in articulating a hardware/software contract—which we call the *unmapped speculation contract*—that captures the above intuition. The contract aims to provide a strong foundation for keeping data confidential, which is typically stated as non-interference. Non-interference can be thought of by considering two system states, s and s', that differ only in sensitive data, which should not be observable by an adversary. A system ensures non-interference if an adversary cannot observe any differences in how the system executes starting from either s or s'.

Single-core USC. To formally state the unmapped speculation contract, we start with a single-core definition. We use $A(\cdot)$ to refer to the state of the CPU, including all architectural and micro-architectural state, but excluding the contents of memory, and we use $M(\cdot)$ to refer to the contents of *mapped* memory, i.e., the contents of every valid virtual address based on the committed page table in that state. We define the contract by considering a single clock cycle of the processor's execution, step$(\cdot)$, which includes any speculative execution done by the processor on that cycle, and require that unmapped pages cannot influence it:

$\forall s, s',$
if $A(s) = A(s')$ and $M(s) = M(s')$,
then with $S := \text{step}(s)$ and $S' := \text{step}(s')$,
it must be that $A(S) = A(S')$

In plain English, the definition considers a pair of starting states s and s' that should look the same, as far as speculative execution is concerned, because they have the same CPU state and the same contents of mapped pages. They might, however, differ in the contents of some unmapped physical pages, which contain sensitive data that we would like to avoid leaking. The definition then considers the state of the CPU at the next clock cycle ($S := \text{step}(s)$ and $S' := \text{step}(s')$ respectively), and requires that the CPU architectural and micro-architectural state $A(\cdot)$, which the adversary might observe, continues to be the same in those two states. As a result, the microarchitectural state could not have been influenced by any sensitive data not present in $M(s)$.

If the OS kernel does not change the mapped memory in that clock cycle, $M(\cdot)$ remains the same, and the contract will continue to hold on the next cycle too. However, if the OS kernel changes the mapped memory, the contract allows speculative execution from that point on to use the newly mapped memory, and the kernel will need to use other mitigations

to defend against transient execution leaks from the newly mapped memory, if necessary.

The contract specifies how the micro-architectural state, $A(\cdot)$, can evolve, but does not say anything about how $M(\cdot)$ can change. This is because the focus of the contract is on transient execution, which cannot affect the committed architectural state of the system; the contents of memory is described by the ISA, since it is architectural state. In other words, changing the memory requires committing the execution of some instruction, at which point this is no longer a transient execution.

Multi-core USC. In a multi-core setting, the CPU state can be thought of as consisting of per-core state (e.g., registers, execution pipeline, and root page table pointer), which we denote with $A_i(\cdot)$ for core i, and the uncore state (e.g., the hardware random number generator [23]), which we denote with $U(\cdot)$, shared by all cores. Similarly, since each core has its own page table, we index the mapped memory by the core i whose page tables we are considering, $M_i(\cdot)$. Finally, we consider the multi-core system executing a clock cycle on one core at a time, $\text{step}_i(\cdot)$. We assume that $\text{step}_i(\cdot)$ does not change $A_j(\cdot)$ for any $i \neq j$. With this notation, the multi-core contract says:

$\forall s, s', i,$
if $A_i(s) = A_i(s'); U(s) = U(s');$ and $M_i(s) = M_i(s'),$
then with $S := \text{step}_i(s)$ and $S' := \text{step}_i(s')$,
it must be that $A_i(S) = A_i(S')$ and $U(S) = U(S')$

This means that speculative execution on core i is allowed to depend on the state of core i, the uncore state, and the memory mapped by core i. This multi-core formulation allows transient execution to affect both the core state $A_i(\cdot)$ as well as the uncore state $U(\cdot)$, at the micro-architectural level. However, transient execution cannot affect either of these states in a way that depends on unmapped memory.

Although hardware threads appear to provide separate execution contexts, with a separate page table for each hardware thread, they have extensive sharing of core resources. To capture that, we consider $A_i(\cdot)$ to include the state of all hardware threads on core i, $\text{step}_i(\cdot)$ to include the execution of any hardware thread on core i, and $M_i(\cdot)$ to be the union of memory mapped by all of the hardware threads on core i (i.e., the union of the page tables of the threads). With this model, the contract allows leakage of mapped memory across hardware threads.

Benefits of the USC. The contract helps reconcile security and performance of speculative execution. On the one hand, hardware can keep the high performance provided by out-of-order execution, because the contract allows almost all forms of speculative execution, as long as data during speculative execution is accessed through non-speculative TLB entries. On the other hand, software can precisely specify what data can and cannot be used for speculative execution, by configuring page tables. For example, if the mapped pages never contain sensitive data, then no mitigations are needed to defend against transient execution vulnerabilities. Finally, because OS developers expect page faults and TLB misses to be quite expensive (compared to memory references), USC doesn't change their performance expectations: developers already have adapted their designs to avoid excessive page faults or TLB invalidations.

Although the contract is aspirational, one appealing property of the contract is that modern computer architectures already effectively aim to provide such a guarantee. AMD explicitly states in bold font that their "processors are designed to not speculate into memory that is not valid in the current virtual address memory range defined by the software defined page tables" [1, pg. 2]. Intel has no explicit position about this contract, but it appears that they treat violations of this contract as bugs to be fixed in hardware or microcode, as evidenced by their fixes for Meltdown and L1TF, described below.

USC and attacks. The contract captures a common pattern that emerges in many transient execution attacks: an adversary can only leak micro-architectural state that is already present on the CPU, as well as the contents of mapped memory, but not the contents of memory that is not present in a page table. As one example, consider the MDS family of attacks [6, 25, 29]. These attacks allow an adversary to trick the kernel into leaking the contents of mapped memory, through careful orchestration of transient execution. Linux prevents this class of attacks by clearing CPU buffers when crossing the user-kernel boundary. This is needed because, when executing in kernel mode, all system memory is mapped and therefore could be leaked through transient execution. The contract, however, captures the fact that only mapped memory is at risk with this attack. This allows for a more efficient mitigation of such attacks, as we demonstrate in WARD, by avoiding kernel mappings of sensitive memory.

In contrast to the example of MDS attacks, which leak sensitive data from memory, the USC does not help mitigate attacks that leak sensitive data already present in the CPU state. For instance, the Spectre variant that leaks the contents of x86 MSRs (Spectre 3a) is not precluded by the contract, since the sensitive data being leaked is not present in memory at all. As a result, an OS kernel must apply other mitigations to deal with such attacks.

More generally, the contract helps categorize existing attacks based on which part of the system state they leak, as shown in Figure 3. For attacks that leak core or uncore state, the contract has little to say in terms of how those attacks can be mitigated, as shown in the "Mitigated by USC" column. As a result, WARD defends against these attacks much in the same way as Linux. In contrast, for attacks that leak the contents of memory, the contract gives a more efficient mitigation approach: simply avoid mapping memory that contains sensitive data. This allows WARD to efficiently mitigate attacks such as some variants of Spectre and MDS.

As shown in the "Consistent with USC" column, all of the

Attack	Leaked state	Mitigated by USC	Consistent with USC
Spectre variants		Yes	Yes
Meltdown		Yes	Yes (ucode)
MDS	Memory	Yes	Yes
PortSmash		Yes	Yes
L1TF		Yes	Yes (ucode)
Spectre variants		No	Yes
LazyFPU	Core state	No	Yes
System reg. read		No	Yes
Spectre variants		No	Yes
CrossTalk	Uncore state	No	Yes
SGAxe		No	Yes

Figure 3: Transient execution attacks categorized based on the state leaked by the attack.

attacks in Figure 3 are consistent with the contract's requirements on the underlying hardware. This is good in two ways. First off, this means that none of the known attacks violate the contract, and thus, the contract is a reasonable approach for mitigating transient execution attacks. Second, this means that USC can mitigate the class of attacks that it covers—namely, attacks that leak memory contents.

There are two special cases: Meltdown and L1TF. When originally discovered, these attacks bypassed the page table protections and allowed an adversary to obtain the contents of memory that was not mapped. In both of these cases, the hardware manufacturer (Intel) considered them to be hardware bugs, as evidenced by the fact that both of them were fixed through hardware and microcode revisions [14, 15], as confirmed by Canella et al. [5].[2]

5 Design

Under the assumption of the unmapped speculation contract, this section describes how WARD can reduce the cost of mitigations for system calls. §5.1 provides an overview of WARD's design with subsequent sections providing more detail about WARD's switch between protection domains (§5.2), about the mitigations used by WARD when mitigations are necessary (§5.3), WARD's kernel text (§5.4), WARD's memory management modifications (§5.5), WARD's process management split (§5.6), and WARD's file system split (§5.7).

5.1 Overview

WARD's design maintains two page tables per process. One page table defines a process-specific view of kernel memory. When a process is running with that page table, we say it is running in its *quasi-visible* domain (or Q domain for short), and with its Q page table. Following the unmapped speculation contract, WARD assumes any kernel memory mapped by the Q page table can be leaked to the currently running process. Instead of using mitigations to prevent leaks of kernel memory, WARD arranges for the mappings in the Q page table to be such that they contain no sensitive data of other processes.

[2]Canella et al. state that some variants of the Meltdown attack, such as Meltdown-BR, are still possible even with the most recent microcode. Those variants, however, are bypassing software checks, rather than the hardware page table, and therefore do not violate the unmapped speculation contract.

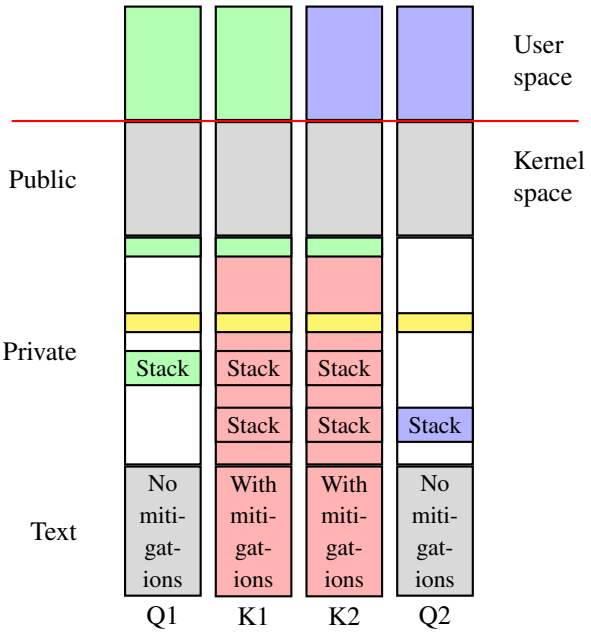


Figure 4: Overview of WARD's address space layout with two processes (indicated by the colors green and purple). Each process has a Q and K domain. Q domains have access to public data (the grey color) and per-process kernel data; the white private region is unmapped kernel data. Each domain also has its own stack and kernel text. In the Q domain, the kernel text has no mitigations. The K domains map all memory, including sensitive memory (indicated by red); all K domains have the same memory layout. Data structures that are shared across processes, such as pipes or file pages, can be mapped in multiple Q domains, as indicated by the yellow color.

When the process needs to access data that is not mapped in the Q page table, it can switch to its other page table, which maps all physical memory, including memory that contains sensitive data. When a process is running with this page table, we say the process is executing in its *K* domain with its K page table. In its K domain, the process runs with the same mitigations as Linux currently uses.

This design allows many system calls to execute in the Q domain, with no mitigation overhead. As a simple example, `getpid` does not access any sensitive data; it needs access to only the kernel text and its own process structure. A more interesting example is mapping anonymous memory: it requires access to the process's own page table and to the memory allocator, but not other processes' page tables or pages.

Figure 4 shows the address space layout in WARD in more detail. Each process has a Q and K view of memory. When a process is running in user space it runs in its Q domain (with no secrets mapped in the Q page table). When a user process makes a system call it enters the kernel but stays in its Q domain. The Q domain maps public kernel memory, Q-visible

kernel memory, the process's Q domain stack, and the kernel text without mitigations.

If a system call needs access to memory in the K domain, WARD performs a switch from its Q domain to its K domain. We refer to the switch from a Q domain to a K domain as a *world switch*, because kernel code in a Q domain runs without most mitigations and the kernel code in the K domain runs with full mitigations. Furthermore, the process switches from its Q domain stack to its K domain stack. The K domain, with access to all kernel memory, can then execute the rest of the system call with full mitigations.

Achieving good performance in WARD depends on avoiding world switches. To reduce the number of world switches, WARD maps kernel data structures that contain no sensitive data into every Q domain. For example, all Q domains map x86 configuration tables (IDT, GDT), some memory allocator state, etc. On the other hand, kernel data structures that contain application data, such as process memory or saved register state, are not mapped into Q domains unless that process should have access to that data.

5.2 World switch

One of the challenges in WARD's design is that a system call often does not know upfront whether it will need to execute in the Q domain or in the K domain. For example, a `read` system call might be able to execute purely in the Q domain, or might need access to sensitive data from the K domain, depending on the file descriptor that the process is reading from, and depending on whether this Q domain already has some sensitive data mapped or not. To support this, WARD's design allows a system call to start executing in the Q domain, and switch to the K domain later as needed.

WARD allows the Q domain to trigger a world switch either *intentionally* or *transparently*. If the code determines that it must switch to the K domain, it can intentionally invoke the function, `kswitch()`, to perform a world switch. When `kswitch()` returns, the kernel thread is now executing in the K domain, and has access to all memory. If the Q domain needs access to specific sensitive data which might or might not be already mapped, the Q domain can attempt to access the virtual address of that data. If the data is already mapped in the Q domain, the access will succeed, no world switch happens, and the Q domain can continue executing. If the data is not mapped, the Q domain triggers a page fault, which transparently triggers a world switch. Once the page fault returns, the kernel thread is now executing in the K domain, as if it called `kswitch()`. Compared to making an intentional call to `kswitch()`, the transparent approach incurs a slight overhead for executing the page fault, but allows large sections of the kernel to be kept completely unmodified, and allows the Q domain to elide a world switch altogether if the data is already mapped in the Q domain.

The above design requires that a kernel thread can start executing in the Q domain and transparently switch to executing in the K domain. This means that any addresses that the kernel thread is referencing, including pointers to data structures, stack addresses, and function pointers, remain the same. To achieve this, WARD ensures that the layout of the Q domain and the K domain match. In particular, all data structures in the Q domain must appear at the same address in the K domain, and the kernel code (text) is located at the same address (even though the code is slightly different, as described in §5.4).

The stack requires particular care because a kernel thread that is processing sensitive data in the K domain could inadvertently write that data to the stack. For example, a `read()` system call from `/dev/random` needs to switch to the K domain to access the system-wide randomness pool. However, the pseudo-random generator code might spill some of its state to the stack, depending on the compiler's choices. If the stack is accessible from the Q domain, the sensitive data could in turn be leaked during the next entry into the Q domain by any thread within the same process. At the same time, if the K domain stack was separate from the Q domain stack, pointers to stack locations before a world switch would no longer work after a world switch. To reconcile these constraints without having to rely on any dedicated compiler support, WARD maps a distinct kernel stack for each domain at the virtual address range and copies the Q domain stack contents to the K domain stack during a world switch.

Transient execution vulnerability	U/Q	K	Ctx
L1TF	x	x	
V1 (Bound Check Bypass)		x	
V1.1 (Bounds Check Bypass Store)		x	
V3 (Meltdown)		x	
V4 (Speculative Store Bypass)		x	
V2 (Branch Target Injection)		x	x
Microarchitectural Data Sampling (Fallout, RIDL, Zombie Load, etc.)		x	x
LazyFPU			x
SpectreRSB			x
PortSmash	Not applicable		
Load Value Injection	Not applicable		
Meltdown-PK (protection key bypass)	Not applicable		
Meltdown-BR (bounds check instr. bypass)	Not applicable		
V1.2 (Read-only Protection Bypass)	Not applicable		

Figure 5: The mitigations implemented in software by WARD.

5.3 Mitigations

Figure 5 shows the known transient execution attacks [5, 12], organized by the mitigations needed to address those attacks in WARD's design. The columns (U/Q, K, and Ctx) indicate where the mitigations are needed: respectively, while executing in user-mode or Q domain; while executing in the K domain; and when context-switching between processes.

The L1TF attack allows leaking the contents of the L1 cache

if there are partially-filled-in entries in the page table. We think of this attack as a violation of the USC (see Figure 3), but a simple microcode fix, as well as clearing unused page table entries, makes the system agree with the USC, and avoids the L1TF attack. Since L1TF allows leaking the contents of any data, WARD applies the mitigations both in user-space, Q domain, and K domain.

The next category of attacks requires no mitigations in either user-space or Q domain. Specifically, Spectre variants that bypass bounds checks require mitigation in the K domain, since there is sensitive memory contents that could be leaked as a result of a speculative check bypass. However, there is no sensitive data that can be leaked in the Q domain, owing to USC. Similarly, no mitigations are required on a context switch, since these attacks can only leak data from the current protection domain.

Meltdown also falls in this category, but for a different reason. Meltdown allows an adversary to bypass the user-kernel boundary check in the page table. WARD's use of a separate page table for the Q and K domains ensures that Meltdown cannot leak any confidential data, since no confidential data is available in the Q domain. Recent microcode from Intel fixes the Meltdown attack in a way that avoids the need for software mitigations.

The next category of attacks require mitigation both in the K domain and on context switch. Spectre v2 and MDS attacks can allow an adversary to obtain sensitive data either from the OS kernel or from another process. However, no mitigations for these attacks are needed in the Q domain due to USC: there is no sensitive data to leak in the Q domain of the currently running process.

For some attacks, such as LazyFPU and SpectreRSB, mitigations are only required on context switch, because the attacks involve process-to-process leakage.

Finally, a number of attacks are not applicable to WARD's simpler design, in contrast to Linux. For example, WARD does not support SGX, does not support running virtual machines, and does not use certain hardware features (such as hardware bounds-check instructions or protection keys).

5.4 Kernel text

Some of the mitigations involve changes to the executable kernel code (text), such as the use of retpolines in place of indirect jumps. These mitigations impose a performance cost, but they are not needed when executing in the Q domain.

A naïve approach might be to compile the kernel code twice, with different compiler flags for mitigations, and load the two different kernel binaries in the Q and K domains respectively. However, this would break WARD's page fault triggered world switches because after completing the switch, execution would resume with the same instruction pointer and stack contents from before the switch but neither would be meaningful in the new text segment.

Instead we need the two version to have matching instruction addresses and stack layouts. WARD achieves this by compiling the kernel only once, but then making two copies of the code at runtime. One copy is mapped into all the K domains, and the other into all the Q domains *but at the same virtual address as in the K domains*. Switching between the two is seamless.

At boot time, in a process inspired by Linux's ALTERNATIVE macro [9], WARD locates each `call` or `jmp` in the Q text segment pointing to a retpoline thunk, and replaces them with the instruction that retpoline emulates. One complication is that indirect call instructions are only 2 or 3 bytes, compared to the 5 that a direct call instruction takes. If we tried to pad with a `NOP` instruction before or after, the pair would not execute atomically, so instead we prepend indirect calls with several repetitions of the CS-segment-override prefix, which is always ignored in 64-bit mode.

5.5 Memory management

Memory allocation in WARD is complicated by the fact that the contents of free pages may contain sensitive data. In particular, if a page was freed by one process, its contents must be erased before the page can be mapped in another Q domain. Zeroing out pages on every allocation would be costly, in particular when allocating kernel data structures, which do not otherwise require the memory to be zero-filled.

To avoid the overhead of repeatedly zeroing kernel pages, WARD implements a sharded allocator for kernel memory. Each Q domain has its own pool of pages for allocation, and the K domain keeps all of the kernel memory that is not part of any Q domain. WARD transfers memory between these shards in batches to amortize the world switch overhead. Keeping a pool of kernel memory in a Q domain allows the kernel to repeatedly allocate and free memory within a Q domain with little overhead.

The other category of memory managed specially by WARD is public memory. WARD maintains a single pool of public pages, with separate functions, `palloc()` and `pfree()`, for allocating and freeing in that pool. All public-pool pages are mapped in every Q domain.

5.6 Process management

When the WARD kernel switches from executing one process to another, it must perform a world switch, to ensure that confidential data does not leak across processes (such as the saved CPU registers that the kernel might save on the stack). However, if a multi-threaded application is running, there is no security reason to perform a world switch when switching between multiple threads in the same process—all of the threads have the same privileges and have access to the same process address space.

To avoid mitigation overhead when switching between threads in the same process, WARD splits the process descriptor, `struct proc`, into two parts. The first part stores sensitive process state, such as the saved CPU registers, and is not public. The second part stores metadata about the process,

such as the PID, the run queue, the scheduler state, etc. This part is public and is used by the scheduler when deciding what thread to execute next. As a result, the scheduler can pick the next thread without incurring a world switch. Furthermore, if the next thread happens to be from the same process, the context switch code can also avoid performing a world switch. Existing scheduler policies that favor picking threads from the same process mesh well with this approach.

5.7 File system

File system workloads involve access to several kernel data structures, including the inode cache and the page cache (containing file data). Inodes are challenging for WARD to deal with because they are smaller than a page, so it is not feasible to map them individually into a Q domain. However, achieving good performance for file system operations requires being able to access an inode without a world switch. To reconcile this conflict, we chose to make all inode structures public in WARD, similar to our approach for splitting the `proc` structure above. If the inode had sensitive data (such as extended attributes), that part of the inode structure would need to be split off into a separate private structure, along the lines of how we split off the part of the `proc` structure storing saved CPU registers.

File data pages are not public, because their contents might be sensitive. WARD implements an optimization that allows it to access file contents without a world switch. In particular, after WARD checks the permissions on a file, it reads or writes the contents of a file page by temporarily mapping the corresponding physical page of memory into its Q domain's address space. This allows the Q domain to access that specific memory page without the risk of leaking other pages; as a result, no mitigations or world switches are needed. When the Q domain is finished with the file read or write, it unmaps the page and issues a TLB shootdown, in case the file is later truncated and the page gets reused for other data.

5.8 Pipes

Pipes are different from many of the other kernel data structures discussed so far in that their contents shouldn't be visible globally, but their state can be associated with multiple processes at a time. WARD's goal is to ensure that if a reader and writer of a pipe run on different cores, then they don't incur world switches when they access the pipe. To achieve this, we store a pipe's data structures in shared memory regions between Q domains. These shared regions are lazily mapped into Q domains the first time a process accesses a pipe (doing the mapping on `fork` would cause unnecessary overhead), and unmapped when the last reference to the pipe within a Q domain is closed.

When a pipe becomes full or empty, the caller blocks on a condition variable. Subsequent reads or writes can observe which processes are blocked and add them to the scheduler run queue if appropriate. Neither of these operations requires access to any secret data so no world switch is triggered until a new process is scheduled. Thus, if the core remains idle until the blocking thread is added back to the run queue, the cost of a world switch is avoided.

5.9 Discussion

WARD's design assumes that there are no secrets in the Q domain that need to be hidden from the user-level process. For many secrets, they can be protected by placing them in the K domain, such as the seed of a system-wide randomness generator. However, address-space layout randomization (ASLR) for the kernel address space is difficult to protect in this fashion, because kernel addresses must be used in the Q domain, and the addresses must match up between the Q domain and the K domain in order for world switches to work. (Note that the initial seed that is used to randomize layout could be protected in the K domain, but the resulting randomized layout cannot be protected.) As a result, kernel ASLR in WARD is susceptible to leakage of addresses through transient execution side-channels.

Our WARD prototype does not include an optimized in-kernel network stack, but a reasonable approach might be to treat all network data as public, leaving it up to the application to encrypt any sensitive information sent over the network. This meshes well with the recent trends in widespread use of TLS for network security, and allows for network operations to achieve high performance in WARD because no mitigations or world switches are required, and all network processing can stay in the Q domain.

Hyperthreading is a source of many possible transient execution leaks, because a significant amount of microarchitectural state is shared between the execution contexts. However, many Linux systems continue to run with hyperthreading *enabled*, despite these risks, because of the high performance overhead they would incur if hyperthreading was entirely disabled. WARD does the same.

6 Implementation

To demonstrate the feasibility of the WARD design, we implemented a prototype of WARD starting from the sv6 research kernel. The kernel is monolithic, implementing traditional OS services such as virtual memory, processes and threads, file systems, fine-grained concurrency using RCU-like techniques, etc. The sv6 kernel, is written in C/C++, runs on x86 processors (both AMD and Intel), and has decent uniprocessor performance and great multicore performance and scalability [8].

Kernel changes. WARD's design affects most core kernel subsystems, including the memory allocator, virtual memory, context switching and the scheduler, and the file system. The simplicity of sv6 allowed for rapid experimentation with kernel designs to enable WARD, which would have been challenging to do in a more complex kernel like Linux, since it is time-consuming to make changes to core subsystems in the Linux kernel, which would have made design iterations far slower.

Transient execution variant	Strategy	Support
V1 (Bound Check Bypass)	bounds clipping	partial
V1.1 (Bounds Check Bypass Store)	lfence	partial
V1.2 (Read-only Protection Bypass)	lfence	n/a (no in-kernel software sandbox)
V2 (Branch Target Injection)	retpoline	yes
—"—	speculation barrier	yes
—"—	return stack buffer filling	yes
—"—	disable spec before BIOS calls	n/a (no calls to BIOS in WARD)
V3 (Meltdown)	Kernel page table isolation (KPTI)	yes
V3a (System Register Read)	microcode	yes
V4 (Speculative Store Bypass)	disable spec. or ctx. switch	yes
LazyFPU	hardware-assisted save/restore	yes
SpectreRSB	return stack buffer filling	yes
L1TF	cache flush, no SMT	n/a (no VM entry in WARD)
—"—	no invalid PTEs	yes
PortSmash	no SMT	no
Microarchitectural Data Sampling	CPU buffer clearing	yes
(Fallout, RIDL, Zombie Load, etc.)	no SMT	no
Load Value Injection	lfence	n/a (no SGX in WARD)
Meltdown-PK (protection key bypass)	address space isolation	n/a (no protection keys)
Meltdown-BR (bounds check instr. bypass)	lfence	n/a (no bounds check instructions)

Figure 6: Transient execution mitigations implemented in WARD.

To help partition the kernel data structures across Q domains, we developed Warden, a tool for tracking down the cause of world switches. Warden instruments page faults from the Q domain that lead to a world switch, and records a stack trace for each of them. Examining the profile of these world switches allows the kernel developer to quickly understand what kernel data structures need to be partitioned or sharded to reduce the number of world switches, as well as the operations that need to be supported on these data structures within a Q domain. Although Warden identifies the data structures that are causing world switches, it is up to the kernel developer to identify an appropriate plan for partitioning the data structure so that no sensitive data can leak through side channels.

To run applications on top of the WARD prototype kernel, we changed the WARD system call interface, including system call numbers, data structure layout, etc, to match that of Linux. This allows unmodified Linux ELF executables to run on top of WARD, and ensures that WARD implements (a subset of) the same system calls that are available on Linux.

We modified sv6 to use PCIDs to reduce the cost of switching page tables (see §5.2). To improve TLB shootdown performance, we modified sv6 to use Linux's shootdown strategy. This is important, for example, for removing temporary mappings in a `read` and `write` systems calls (see §5.7).

Mitigations. WARD implements side-channel mitigations for known transient execution attacks [5, 12], as shown in Figure 6. WARD mostly copies the mitigation strategies and their implementation from the Linux kernel [19]; the most interesting exception is that WARD does not apply some of these mitigations to the Q domain, as described in Figure 5.

For Spectre V1, WARD, adds an `lfence` instruction when copying from user code, and when taking an interrupt, exception, and NMI entry. WARD uses bounds clipping in fewer cases than Linux for two reasons: WARD has less code and we haven't performed a careful audit of the complete source code. For Spectre V2, we compile WARD to use retpolines (by specifying the "-mretpoline-external-thunk" flag to `clang`). WARD also uses Linux's `FILL_RETURN_BUFFER` macro to fill the return stack buffer, and issues an indirect branch predictor barrier `IBPB` instruction on a context switch. For Spectre V3, WARD uses separate page tables (as described in §5.1) and uses process-context identifiers (`PCID`s) to avoid TLB flushes.

For Spectre V4, WARD issues an `lfence` on context switch. (If WARD supported generating code at runtime, the JITs would also have to be hardened.) For LazyFPU, WARD uses the `xsaveopt` instruction to safe/restore floating point state. For SpectreRSB, WARD fills the return stack buffer on context switch. For L1TF, WARD avoids invalid PTEs. Like Linux, WARD doesn't address PortSmash; the default for the Linux kernel is to allow SMT, and WARD does too. For microarchitectural data sampling attacks, WARD issues the `verw` instruction for clearing CPU buffers.

Some attacks aren't applicable to WARD, because WARD doesn't support virtualization, secure enclaves, and hardware transactional memory; does not call into the BIOS; and does not implement in-kernel software sandboxes such as BPF.

Like Linux, WARD also zeroes unused CPU registers on kernel entry, to reduce the avenues of attack available to an adversary. To determine whether mitigations are necessary, WARD maintains a special variable called `secrets_mapped` whose value is 0 in the Q domain and 1 in the K domain; this allows the rest of the kernel code to determine if it needs to perform mitigations just by using `if (secrets_mapped)`

... (as long as interrupts are disabled, to avoid races). To help evaluate the performance impact of side-channel mitigations, WARD's implementation allows switching individual mitigations on and off at runtime, rather than at compile time or boot time.

To improve performance, a few system calls invoke the world switch intentionally to avoid the extra overhead of a transparent world switch. For example, `open`, and `fork` always invoke world switch intentionally. The `read` and `write` system calls invoke a world switch intentionally when they are reading or writing large amounts of data, since the cost of a world switch is less than the cost of shooting down the temporary mappings for that many file pages. A page fault on a Copy-On-Write (COW) page also intentionally invokes a world switch.

Lines of code. The WARD prototype consists of about 34,000 lines of C++ code (for `kernel/` and `include/`), compared to 24,000 lines of C++ code for the sv6 kernel that WARD was derived from. `git diff -stat` reports roughly 17,000 lines of insertions and 5,000 lines of deletions between sv6 and WARD. It is difficult to further break down WARD's lines of code, since many aspects of WARD's design required small changes throughout the kernel's source code. For example, splitting up the kernel memory allocator required the use of C++ placement `new` in many parts of the kernel. Similarly, implementing the Linux binary compatibility layer required making changes to the implementation of many system calls.

7 Evaluation

To demonstrate the benefits of WARD's design, this section answers the following questions:

- Do WARD's techniques reduce the overhead of mitigations for system calls? (§7.2)
- How do mitigations affect the cost of a world switch? (§7.3)
- What are the memory overhead associated with WARD's design? (§7.4)

7.1 Experimental methodology

To answer these questions, we consider three different configurations of WARD:

- Baseline: WARD with no mitigations against side channels.
- Linux-style: WARD with standard mitigations against side channels, mirroring the approach taken by the Linux kernel. This configuration does not use separate Q domains; all system calls directly enter the K domain.
- USC-based: WARD with fast mitigations that take advantage of the split between the Q domain and the K domain, leveraging the USC. The K domain implements the same mitigations as in Linux-style.

WARD's design is aimed at reducing the overhead of mitigations associated with system calls. To zoom in on the system call overhead, we evaluate WARD's performance using LEBench [24], a collection of system call workloads representative of a range of real applications. This allows us to precisely report and explain the effect of WARD's techniques on individual system calls. We don't report results for the networking benchmarks in LEBench, because the WARD prototype doesn't have a suitable in-kernel network stack.

All benchmarks were run on a Dell PowerEdge T430 with two E5-2640 v4 CPUs and 64 GB of RAM.

One potential concern with the use of recent microcode is that it makes the baseline slower, which in turn makes the cost of mitigations appear lower than they really are. This is similar to the significant effect we observed with newer CPUs, as described in §2. However, with newer microcode, we find that the performance of the baseline is not significantly affected: it achieves similar performance even when we use old microcode. The reason for this is that the recent microcode updates add mitigations that can be specifically enabled (e.g., through the `SPEC_CTRL` MSR), but almost nothing is enabled by default. The Linux and Ward baseline experiments do not enable these mitigations, and thus the performance effect is minimal.

For the Linux measurements of LEBench, we use the 5.4.0 kernel on Ubuntu 20.04.

7.2 WARD's USC-based fast mitigations

LEBench. Figure 7 shows the benefit of WARD's fast mitigations on LEBench. The figure compares WARD with USC-based and Linux-style mitigations, relative to the baseline with no mitigations. As shown, WARD with fast USC-based mitigations is often able to match the unmitigated baseline. The reason is that many of the microbenchmarks can execute with no or very few world switches, as shown in Figure 8.

Many microbenchmarks (`getpid` through `huge pagefault` in Figure 8) have nearly 0 transparent and intentional world switches. They execute completely in the Q domain. The reason that some have near 0 world switches, but not exactly 0, is that during the measurement they were interrupted by a timer interrupt, which requires a world switch to the K domain to run the scheduler (the remainder of the syscall is then executed in the K domain too).

Another cause for fractional numbers of transparent world barriers is that some operations might have a slow path that requires secrets but only gets triggered infrequently (i.e. because a memory allocator pool ran empty). A strength of the WARD approach is that these sorts of cases don't have to be manually annotated and in fact it is harmless to completely ignore them provided they are executed infrequently enough.

There are several microbenchmarks (e.g., the bigger `read` and `write` ones) that perform one intentional world switch per system call. These system calls immediately enter the K domain and thus perform identical to WARD with full mitiga-

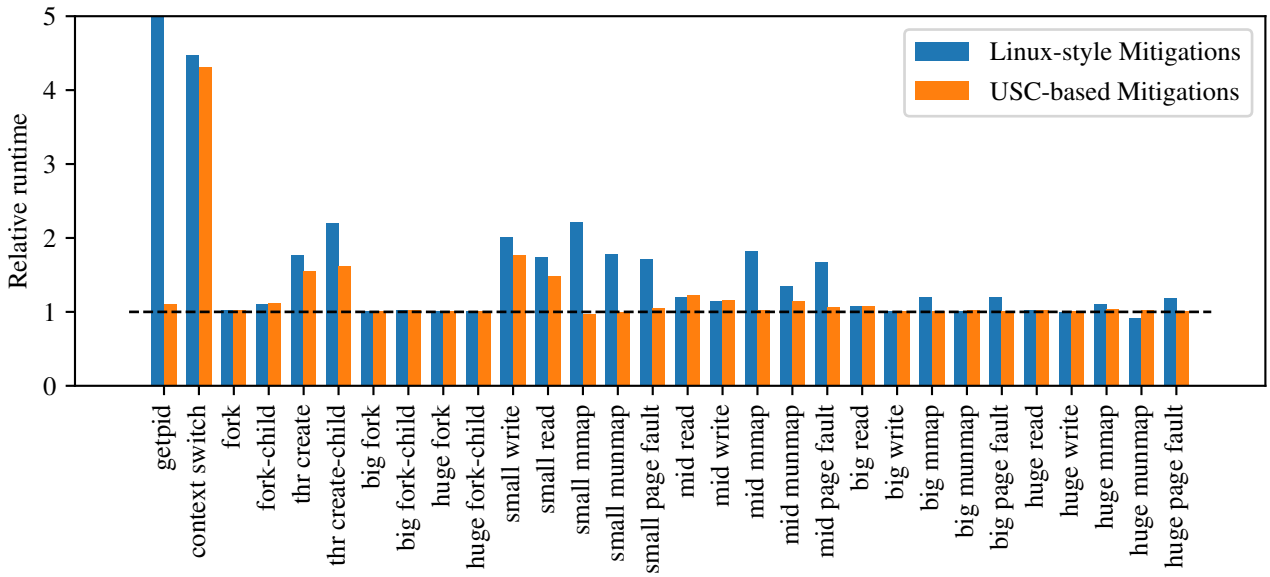


Figure 7: Performance of WARD with fast USC-based mitigations and with Linux-style mitigations, normalized against the baseline performance of WARD without any mitigations.

tions, and have the same overhead. These system calls also perform much work in the kernel and the overhead of the 1 world switch is amortized by that work.

The `thr create` and `thr create-child` do multiple syscalls per iteration, but average one world barrier per iteration. Specifically, the `thr create` microbenchmark makes three systems calls: one `clone` that requires a world switch and a call to each of `sigprocmask` and `set_robust_list` which don't. The `thr create-child` microbenchmark includes an additional call to (`sigprocmask`) from the child process, for which WARD can also avoid the world switch.

The `fork` and `fork-child` benchmarks each do a single syscall with an intentional world barrier that takes the vast majority of execution time, but also raise a handful of page faults to populate page table entries (which need secrets if they are copy-on-write related or if the kernel runs out of zeroed memory pages and has to prepare more).

An interesting case is the `context switch` microbenchmark. This microbenchmark measures context switching by writing and reading a byte over a pipe between two processed pinned to the *same* core. The `write` calls avoids a world switch because the scheduler can wake other processes while in the Q domain, but the `read` call causes a context switch and (since the two processes are mutually distrusting) thus requires a world switch.

When we modify the microbenchmark to pin the two processes to *different* cores we observe that it runs without world switches and that the overhead is about 25 times lower than Linux-style mitigations.

Application: git. To confirm that the improved performance of WARD's fast mitigations seen in LEBench translates into

	# sys calls	World switches		
		T	I	Sum
getpid	1	0	0	0
small write	1	0	0	0
small read	1	0	0	0
small mmap	1	0	0	0
small munmap	1	0	0	0
small page fault	1	0	0	0
mid mmap	1	0	0	0
mid munmap	1	0	0	0
mid page fault	1	0	0	0
big mmap	1	0	0	0
big page fault	1	0	0	0
huge mmap	1	0	0	0
huge page fault	1	0	0	0
context switch	2	0	1	1
thr create	3	0	1	1
thr create-child	4	0	1	1
mid read	1	0	1	1
mid write	1	0	1	1
big read	1	0	1	1
big write	1	0	1	1
big munmap	1	1	0	1
huge read	1	0	1	1
huge write	1	0	1	1
huge munmap	1	1.001	0	1.001
fork	2	0	2	2
big fork	2	0	2	2
huge fork	2	0	2	2
huge fork-child	17	0	7	7
big fork-child	17	0.006	7.02	7.026
fork-child	17	0.012	7.065	7.077

Figure 8: The microbenchmarks, sorted by the sum of the number of transparent (**T**) and intentional (**I**) world switches per iteration, along with the number of system calls invoked (including page faults).

Configuration	Transparent	Intentional
None	2457 cycles	1082 cycles
SpectreV2	2453 cycles	1075 cycles
MDS	3337 cycles	1980 cycles
MDS+SpectreV2	3363 cycles	1992 cycles
MDS+SpectreV2+Q_retpoline	3406 cycles	2014 cycles

Figure 9: The costs of transparent and intentional world switches for different configurations.

application-level performance improvements, we evaluated the performance of `git`. For this benchmark, we ran `git status` in a 100 MB repository that we cloned from GitHub; all of the file system state was cached in memory. The average runtime for Linux-style mitigations took 24.6% longer than the unmitigated baseline, and USC-style mitigations took 11.2% longer than the unmitigated baseline. Much of the speedup is due to the fact that `git status` invokes frequent `lstat` system calls, which can execute in the Q domain. The remaining overhead is due to system calls like `openat` that require a world barrier for accessing potentially sensitive file contents.

7.3 World switch

§7.2 shows that the mitigation overhead is dominated by the cost of a world switch. This section breaks down this cost.

An intentional world switch via `kswitch()` takes around 644 cycles on a shallow stack, plus 50 cycles or so for every KB of stack used (the cost of a `memcpy`). A transparent world switch using a page fault adds 1372 cycles.

Figure 9 measures the cost of a null system call that invokes an intentional or a transparent world switch, and returns. It shows the cost for different configurations: no mitigations, MDS mitigations, SpectreV2 mitigations, and with `retpoline` in Q domain. The configuration with Q_retpolines runs with retpolines in both the Q and K domains. It shows the benefit of WARD patching them out at runtime: the retpoline that disables branch prediction for indirect jumps through the system call table costs 22 cycles.

7.4 WARD memory overhead

Because the memory protection mechanisms that WARD uses to expose non-secret data to Q domains operates on a 4KB or 2MB granularity, WARD's approach incurs some additional memory overhead. Figure 10 lists some of these cases. In general we face a trade-off when filling small dynamic memory allocations for Q domain state: either we use an entire page each time, or we tolerate higher memory fragmentation because all chucks of memory on a page must be only used by the same Q domain.

7.5 Security

To validate that WARD's mitigations work, we implemented a demonstration program that attempts to execute a Spectre V2 attack against the WARD kernel. While running with applicable mitigations disabled (i.e. each Q and K domain retpoline replaced with a normal indirect jump) the attack succeeds in exfiltrating secret kernel data. However, when our Spectre V2 mitigations are re-enabled (by re-enabling retpolines in the K domain) the attack is thwarted. It is of course impossible to be certain that all variations on the attack would be blocked, but this test provides some confidence both that the unmitigated baseline is vulnerable to transient execution attacks, and that WARD is able to prevent them.

Component	Overhead	Explanation
Kernel text	2 MB	Separate text segments for Q and K domains
Public kernel data	< 4 KB	Padding to a page boundary
Process structure	4 KB / process	Allocated on its own page
Thread structure	~6 KB / thread	Split between a Q domain page and a K domain page
Q domain stack	32 KB / thread	Smaller stacks possible by avoiding deep recursion
Page tables	*varies*	Q domain mappings require additional PTEs
Inodes	–	Many public allocations
Scheduler state	–	packed into a single page

Figure 10: Memory overhead of different WARD components.

8 Discussion

Future vulnerabilities. It is likely that there are further transient execution attacks either under embargo or yet to be discovered. Based on trends in the existing attacks, we believe that WARD should be well positioned to address them: so far, mitigations developed for Linux have been suitable to directly copy into WARD. Since many need to run only at K domain entry/exit instead of every user-kernel boundary crossing, the same defenses in WARD might be cheaper to apply than they would be for Linux.

Linux. We are optimistic that Ward's techniques could also benefit monolithic production kernels for two reasons. First, WARD and Linux are in the same ballpark in terms of system call performance on LEBench. Out of the 30 microbenchmarks, WARD is faster than Linux on 18 of them, and slower on 12. Second, as shown in Figure 1 (§2) Linux incurs a significant overhead for mitigations on LEBench and that overhead is in line with the overhead that WARD's Linux-style mitigations incur on LEBench (see Figure 7). Some systems calls experience more overhead in WARD, because they implement less functionality (e.g., `getpid`), but the corresponding calls in Linux also incur significant overhead. Some systems calls in WARD have less overhead than Linux, because they are not as efficient; for example, big and huge `mmap` in WARD requires an update of its radix-tree VM data structures [7], while Linux just inserts the new region into a list. Linux may see a bigger pay for those system calls with WARD's design than WARD.

A question is how much effort is required to incorporate WARD's techniques into a production kernel such as Linux. Our preliminary efforts have proven encouraging: we found that we could leverage existing infrastructure for KPTI to

maintain Q domain and K domain page tables. We implemented a `switch_world` function in Linux, which switches to the K domain and copies the Q stack to the K stack. We modified the Linux page-fault handler to call this function when it encounters a page fault while running with the Q page table. This allows the Linux kernel to run as normally with a transparent world switch on each system call. We refactored the `struct task_struct` into a Q-private and secret part, allowing the `gettid` system call to run completely in the Q domain. This gives us some indication that the basic approach of WARD could be made to work in Linux, although an open question is how to best re-design the data structures in the Linux kernel to fit WARD's design.

9 Related work

This paper is motivated by the papers that show how secret kernel data can be leaked through micro-architectural state (e.g., [4, 6, 16, 21, 25, 29]). In particular, two survey papers were helpful by categorizing the known attacks [5, 12].

This paper relies heavily on the mitigation work in the Linux community [19]. WARD adopts Linux's techniques and their optimized implementation in the K domain. WARD uses, for example, Linux's `nospec` macro for bounds clipping, `FILL_RETURN_BUFFER` to fill the return buffer, and retpoline. WARD's hotpatching of its kernel text to remove retpolines in the Q domain was inspired by Linux's ALTERNATIVE macro [9].

In addition to the software/microcode approach currently used by Linux and other production operating systems, there are several proposed hardware-only defenses that delay the use of speculative data until it is safe [3, 31, 34]. While these defenses are more comprehensive, they have higher overheads that impact performance whenever speculation occurs. By contrast, the USC constrains speculation in a more targeted way based on memory mappings. ConTExT also proposes constraining speculation based on memory mappings, but introduces a new PTE bit to explicitly mark pages that contain secret data [26]. WARD instead keeps secrets in separate address spaces, and allows speculation after employing its defenses to switch to the K domain. Finally, SpecCFI proposes to enforce control-flow integrity during speculative execution [18]. This idea strengthens Spectre defenses, and is complementary to WARD.

The Q page table is inspired by the shadow page table in KAISER [11] and KPTI [20]. In Linux, when a process executes in user space, the process runs with a shadow page table, which maps only minimal parts of kernel memory: the kernel memory to enter/exit the kernel on a system call. As soon as the process enters the kernel, it switches to the kernel page table that maps all of physical memory. WARD, however, executes complete system calls while running under the Q page table; this requires a significant redesign of the OS kernel, which is a major focus of this paper.

The use of virtual-memory to partition the kernel address space has a long history in operating systems research. One example is Nooks [27], which runs device drivers in separate protection domains with their own page table in kernel space to provide fault isolation between drivers and the kernel. Another example is the use of Mondrian Memory Protection [32] to isolate Linux kernel modules in different protection domains within the kernel address space [33]. The most recent example is Mike Rapoport's work on kernel address space isolation [10] in Linux. These designs use similar techniques to introduce isolation domains within the kernel, but focus on traditional attacks (e.g., code execution through a buffer overflow) as opposed to transient execution.

10 Conclusion

This paper articulates the unmapped speculation contract (USC) for a division of labor between hardware and software. This contract allows hardware to speculate on many values (but not the values of page table entries) and provides software with a mechanism to prevent leaking secrets through micro-architectural state. The WARD design shows how USC can be used to reduce the performance costs of mitigations on system calls using per-process Q domains and global K domains. WARD transparently switches from Q- to K-domain through page faults, uses temporary mappings to access unmapped physical pages, and splits data structures into public and private parts. An evaluation shows that WARD can run the microbenchmarks of LEBench with small performance overhead compared to a kernel without mitigations: for 18 out of 30 LEBench microbenchmarks, WARD's performance is within 5% of the performance without mitigations . Although WARD is research kernel, we are hopeful that its ideas can carry over to production monolithic kernels.

Acknowledgments

We'd like to thank the anonymous reviewers and our shepherd, Chris Hawblitzel, who provided comments that helped improve this paper. We also want to thank our artifact evaluators for their diligent examination of our artifact submission.

Artifact

Source code and directions for using WARD are available at `https://github.com/mit-pdos/ward`.

References

[1] Advanced Micro Devices, Inc. Speculation behavior in AMD micro-architectures. `https://www.amd.com/system/files/documents/security-whitepaper.pdf`, 2019.

[2] Apple, Inc. Additional mitigations for speculative execution vulnerabilities in Intel CPUs. `https://support.apple.com/en-us/HT210107`, August 2019.

[3] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, pages 151–164, Seattle, WA, September 2019.

[4] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, pages 991–1008, Baltimore, MD, August 2018.

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *CoRR*, abs/1811.05441, 2018.

[6] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pages 769–784, London, United Kingdom, November 2019.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM EuroSys Conference*, pages 211–224, Prague, Czech Republic, April 2013.

[8] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, November 2013.

[9] Jonathan Corbet. SMP alternatives. `https://lwn.net/Articles/164121/`, 2005.

[10] Jonathan Corbet. Generalizing address-space isolation. `https://lwn.net/Articles/803823/`, November 2019.

[11] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: Long live KASLR. In *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems*, pages 161–176, Bonn, Germany, July 2017.

[12] Mark D. Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L. Hennessy. On the Spectre and Meltdown processor security vulnerabilities. *IEEE Micro*, 39(2):9–19, 2019.

[13] Intel, Inc. Deep dive: Retpoline: A branch target injection mitigation. `https://software.intel.com/security-software-guidance/deep-dives/deep-dive-retpoline-branch-target-injection-mitigation`.

[14] Intel, Inc. Software guidance: L1 terminal fault. `https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault`, 2018.

[15] Intel, Inc. Software guidance: Rogue data cache load. `https://software.intel.com/security-software-guidance/software-guidance/rogue-data-cache-load`, 2018.

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 19–37, San Francisco, CA, May 2019.

[17] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO)*, pages 104–113, Santa Barbara, CA, August 1996.

[18] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SpecCFI: Mitigating Spectre attacks using CFI informed speculation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 39–53, San Francisco, CA, May 2020.

[19] Linux Kernel Maintainers. Hardware vulnerabilities. `https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/`, 2020.

[20] Linux Kernel Maintainers. Page table isolation. `https://www.kernel.org/doc/Documentation/x86/pti.txt`, 2020.

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, pages 973–990, Baltimore, MD, August 2018.

[22] Microsoft Corporation. Windows guidance to protect against speculative execution side-channel vulnerabilities. https://support.microsoft.com/en-us/help/4457951/, November 2019.

[23] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores area real. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2021.

[24] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 554–569, Huntsville, Ontario, Canada, October 2019.

[25] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pages 753–768, London, United Kingdom, November 2019.

[26] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. Context: Leakage-free transient execution. *CoRR*, abs/1905.09100, 2019.

[27] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), November 2004.

[28] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxe.com, 2020.

[29] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 88–105, San Francisco, CA, May 2019.

[30] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM EuroSys Conference*, pages 18:1–18:17, Bordeaux, France, April 2015.

[31] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*, pages 572–586, Columbus, OH, October 2019.

[32] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, San Jose, CA, October 2002.

[33] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 31–44, Brighton, United Kingdom, October 2005.

[34] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*, pages 954–968, Columbus, OH, October 2019.

[35] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM EuroSys Conference*, pages 379–391, Prague, Czech Republic, April 2013.

Predictive and Adaptive Failure Mitigation to Avert Production Cloud VM Interruptions

Sebastien Levy[†], Randolph Yao[†], Youjiang Wu[†], Yingnong Dang[†], Peng Huang[◇]
Zheng Mu[†], Pu Zhao[⋆], Tarun Ramani[†], Naga Govindaraju[†], Xukun Li[†]
Qingwei Lin[⋆], Gil Lapid Shafriri[†], Murali Chintalapati[†]

[†]*Microsoft Azure* [◇]*Johns Hopkins University* [⋆]*Microsoft Research*

Abstract

When a failure occurs in production systems, the highest priority is to quickly mitigate it. Despite its importance, failure mitigation is done in a reactive and *ad-hoc* way: taking some fixed actions only after a severe symptom is observed. For cloud systems, such a strategy is inadequate. In this paper, we propose a preventive and adaptive failure mitigation service, NARYA, that is integrated in a production cloud, Microsoft Azure's compute platform. Narya *predicts* imminent host failures based on multi-layer system signals and then decides smart mitigation actions. The goal is to *avert* VM failures. Narya's decision engine takes a novel online experimentation approach to continually explore the best mitigation action. Narya further enhances the adaptive decision capability through reinforcement learning. Narya has been running in production for 15 months. It on average reduces VM interruptions by 26% compared to the previous static strategy.

1 Introduction

Failures are common in large systems. High-availability system designs require techniques that address a key question: *once a failure occurs, how to quickly detect and mitigate it so the system can continue running?* Mitigating a failure here means attempting to make the failure symptom disappear without necessarily diagnosing and fixing the underlying bugs first. However, for a large cloud infrastructure like Microsoft Azure that serves millions of customers running virtual machines and various software atop, only employing post-failure detection and mitigation techniques is insufficient.

This is because if a system only takes mitigation actions after a failure is detected, users may already be having bad service experience as the system runs in a degraded mode (not completely failing) [15, 17, 29]. Moreover, when a failure is detected, the system will be under intense pressure to mitigate the failure fast in order to minimize downtime; but in practice failure mitigation takes time for large systems, and expediting mitigation could even worsen the situation [12]. In addition, our experience suggests that even short, mitigated failures can be impactful to customers due to the interruptions themselves.

Therefore, cloud systems should also design techniques to address the question of, *whether a failure may be imminent, and if so, what preventive actions should be taken to* ***avert*** *this failure?* Several recent works tackle the failure prediction problem [14, 27, 38] in the context of disk failures. But they focus on prediction alone, with the goal of alerting operators or providing allocation hints [25]. The questions of how much benefit does the prediction bring, and more importantly what preventive mitigation actions should the system take in response to predicted failures remain open. Answering these questions requires a holistic solution—one that is closely integrated in the system's control loop, which not only predicts host failures in real time, but also automatically decides the proper mitigation actions, measures the benefits, and continuously adjusts its actions based on the measured benefits.

In this paper, we present NARYA to fill this aforementioned gap. Narya is an end-to-end service with predictive and smart failure mitigation fully integrated in the Azure compute platform for its Virtual Machine (VM) host environment. The design goal of Narya is to *prevent* VM failures ahead of time and enhance the self-managing capability of the Azure compute platform for providing smooth VM experience to customers.

Narya's design is informed by several observations we had. First, while failure mitigation is a crucial step in cloud operation, the current practice is *ad-hoc*. To mitigate a (predicted) failure, developers use static policies that prescribe actions based on the symptoms and domain knowledge. While this approach works for simple systems, it does not work well at Azure scale. With multi-tenancy, heterogeneous infrastructure components, and diverse customer workloads, it is difficult to comprehensively categorize different failure scenarios in a large cloud system beforehand and determine good mitigation actions (or their parameters), especially without trying it.

Moreover, as the cloud system is constantly changing (software/hardware updates, customer workload changes), some mitigation action that worked well in the past may no longer be optimal. As a result, developers keep reactively adjusting the actions based on hind sights from service incidents.

For example, initially restarting a host node upon receiving a predictive failure signal may be effective as the system failures tend to be caused by some transient hardware issues; but gradually permanent node failures become more common so restarting is not the best mitigation action anymore—live migrating the virtual machines from the node predicted to fail to a healthy node may be a better action. Therefore, for cloud-scale systems, we need smart and adaptive failure mitigation.

Our insight is that the effectiveness of taking some mitigation action in a complex and changing system is often probabilistic as there are too many factors affecting it (network condition, VM size, applications, hardware health, customer activities, etc.), which may not be thoroughly accessed or assessed. We usually do not know beforehand whether some mitigation action is good or not, or whether there is a better action, unless we try it. Consequently, explorations with *production* workload is indispensable to determine the (near-)optimal failure mitigation action. Nevertheless, we should ensure that the actions taken maximize the expected effectiveness (minimize the potential customer impact) over time.

Based on this insight, Narya takes a novel online experimentation approach. In particular, Narya predicts whether host nodes in the production fleet will likely fail and then leverages A/B testing to continually experiment with different mitigation actions, measure the benefits, and discover optimal actions. The rationale behind the A/B testing strategy is that it, in essence, introduces randomization that avoids biased pivots of the diverse nodes, which helps surface the statistically significant effective actions.

One important drawback of the A/B testing strategy is its cost of exploring each action until statistical significance is found and then always choosing the estimated best action. This problem is essentially the classic *exploration–exploitation* trade-off [32] in learning systems that need to make decisions with incomplete information (about the system stack, customer workloads, etc.), constant changes, and uncertain pay-offs (whether the action will prevent future failures). The dilemma is whether the learning agent should repeat a mitigation action that has worked well so far, *i.e.*, *exploit*, or it should try some novel choices in the hope of getting better rewards, *i.e.*, *explore*. We address this issue by enhancing the Narya decision engine using a dynamic assignment learnt through a multi-armed Bandit model [2, 33]. This helps decrease overall cost by better leveraging the early cost estimation of each action and by continuously exploring each of them to adapt to system changes.

Narya has been running in production for 15 months in Azure as part of the Gandalf [24] suite. Narya successful prevents many VM interruptions for customers. In nine production experiments that Narya runs for different failure types, Narya on average reduces VM interruptions by 26% compared to the previous static strategy. This reduction is close to what the oracle optimal strategy could achieve (35%).

The major contributions of this work are:

- We propose a holistic failure avoidance solution that includes failure prediction, new failure mitigation actions, and intelligent mitigation strategies.
- We design a novel approach of using A/B testing for online experimentation with production workload to automatically identify good failure mitigation actions.
- We explore a more advanced reinforcement learning approach to optimize choice of mitigation action.
- We evaluate the proposed solution in a large-scale, production cloud service, Azure, to validate its effectiveness.

2 Background and Motivation

A traditional system's operation cycle is as follows: a failure is detected; developers diagnose the failure and find out the root cause; a patch is written; the system is re-deployed. For cloud systems, operating in this exact sequence is problematic because the time it takes to identify the root cause and develop a fix is usually long and exceeds the downtime budget. Instead, once a failure is detected, some *mitigation* action like restart will be applied first without necessarily knowing the bug.

2.1 Target System and Goal

We tackle the problem of preventive and smart failure mitigation for cloud systems. Our specific target system is the VM host environment, a node, in the Azure compute platform. The host environment is a complex stack consisting of guest OSes, guest agents, hypervisor, host OS, host agents, firmware, and hardware. The node is backed by locally attached disks and remote virtual disks. Each node is connected to various compute services, together referred to as controller, that is responsible for provisioning resources and performing management actions such as creating and destroying VMs.

Azure already employs layers of monitoring mechanisms to actively detect if a host node has failed (e.g., via periodic pings), and mitigate the failure with actions such as rebooting the node. We aim to further develop techniques that *predicts* whether a host environment might fail soon and *automatically* decides an appropriate mitigation plan among multiple choices. The end goal is to *avoid* future VM failure events.

2.2 Are Failures Predictable?

To predict failures, there are two basic requirements: (i) the imminent failure is not abrupt; and (ii) there is telemetry recorded to indicate the degradation. One type of predictable hardware issue is certain hardware parts wear out. We could predict using the age or the wear-out rate. Combined with other system signals such as workload patterns, we can predict if a host will fail soon. Resource leak, including memory/file handle/network ports leak, is a common type of predictable software failure. We could predict them using the resource usage trend. If failures are correlated with certain hidden factors such as timeout settings, bugs related to timers, and release schedule, they may also occur on a predictable basis.

Timestamp	Event
03-14 18:00:00	A node with 16 VMs was predicted to fail with 0.7 prob.
03-16 01:09:12	Node agent crashed, 17 VMs offline
03-16 01:15:26	Controller probes to node agent timed out, retry
03-16 01:31:00	Node state marked by controller as unhealthy
03-16 02:07:37	Controller tried to recover the node through restart
03-16 02:23:02	Failed to receive node reboot success signal
03-16 02:23:33	VMs in the node were recreated in another node
03-16 04:40:17	Node was sent out for repair
03-16 06:13:21	Offline diagnosis finished, disk fault was suspected

Table 1: Events timeline for a production node.

2.3 Why Reacting on Predicted Failure?

Since predicted failure is about something (complete failure) that has not occurred yet, one option is to only treat it as an early warning and not act on it. After all, it is developers' common mindset that *"If It Ain't Broke, Don't Fix It"*. However, for cloud services, this mindset puts customers and their VMs at the risk of suffering interruptions. With techniques such as live migration which can migrate a VM from one node to another with minimal customer impact, cloud vendor is in a better position to help customer avoid failures.

To give an example, Table 1 shows a production case of the timeline for events in a node. In this case, the node was predicted to fail with a relatively high probability, but no preventive action was taken. So both the existing VMs and new VMs still run in this node. Later, this node indeed failed (OS crash), which caused 17 VMs including 1 new VM to be offline. The controller tried to probe the VMs and timed out. Then it tried to reboot the node but failed. Finally, the controller decided to recreate the VMs in another node. Subsequent offline diagnosis confirmed the disk on the node was indeed problematic. Had we taken some mitigation action when receiving the failure prediction signal, we could have saved the long VM disruptions and customer impact.

2.4 Why Static Mitigation Is Insufficient?

Intuitively each predicted failure should be handled in the same way using an optimal method. Indeed, initially we used a static strategy where all predicted bad nodes would be mitigated using the same plan: 1) block allocation on the node; 2) try to live migrate VMs; 3) wait for 7 days for short-lived VMs to be destroyed by customers; 4) force migration of remaining VMs; 5) mark the node offline and send it for repair. Although this plan looks reasonable, it quickly faces limitations. Blocking allocation results in capacity pressure while for some predicted failures, avoiding allocation may be better. Some failures may be too severe to do live migration (*e.g.*, broken disks). Forced migration causes unnecessary customer impact if nodes are still healthy after 7 days. Marking nodes offline is also suboptimal when capacity is low.

Using static assignment prevents us from knowing what would have happened if we had chosen a different action, and therefore from knowing how much customer pain we saved or if we were using the best action. In addition, static

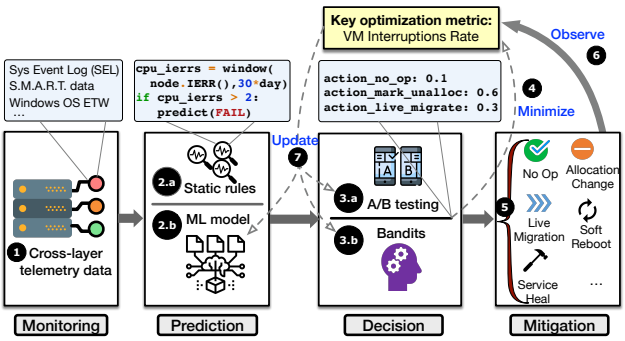


Figure 1: High-level workflow of Narya.

mitigation get affected by system changes over time. With the complexity of our cloud system, the effect of a rule, as well as the telemetry that a rule relies on, is constantly changing. In these cases, since static mitigation does not try other actions, it could increase customer pain without us realizing it. For example, a low CPU frequency can be a defense mechanism from a CPU to indicate an imminent failure; detecting such drop in CPU frequency and applying a mitigation action can be beneficial. However, new improvements in the system could voluntarily decrease CPU frequency to conserve energy. In this case, the original rule could have a high number of false positive in prediction results, and applying the same action will very likely cause more harm than good.

Another limitation of static failure mitigation is that it creates tendency for developers to make *ad-hoc* modifications to the mitigation assignment based on some isolated cases. No mitigation action can be perfect and customers may complain if they suffer from such mitigation. In such case, developers tend to modify the rule to satisfy the customers. However, without testing if that change does reduce the *overall* customer pain, it is possible that the situation gets worse and the policy might be switched back again when another customer complains about this new policy.

3 Overview

We design Narya, an end-to-end service that is integrated in the Azure compute platform, to predict host failures and automatically decide what mitigation actions to take for each predicted failure. The design goal of Narya is to *avert* potential VM failures while minimizing the impact to customers. Narya advances the current practice of failure mitigation in two ways: (i) replacing existing static and *ad-hoc* mitigation assignment to adaptive and systematic decision algorithms; and (ii) transforming the traditionally reactive, post-failure mitigation activity to proactive failure avoidance mechanisms.

3.1 Narya Workflow

Narya takes a novel online experimentation and learning approach to the failure mitigation problem. Figure 1 shows overview of the high-level workflow in Narya.

Each node in Azure is deployed with monitoring agents that

collect various telemetry signals about the host environment (❶, §4.1). The prediction component in Narya continuously consumes these signals and predicts whether some node will fail soon (❷). The prediction is made by both static domain rules (2.a, §4.2) and running machine learning inference (2.b, §4.3). Each prediction result is streamed into the decision component in Narya as a mitigation request. Narya supports two decision schemes: A/B testing (3.a, §6.1), and Bandit model (3.b, §6.2). The decision component computes a probability distribution of applicable mitigation action choices (❹, §5) and then picks an action based on the distribution. The mitigation controller applies the chosen mitigation action to the suspected node (❺). This whole process is an automated feedback loop that optimizes a key objective metric—VM interruption rate (§3.2). Narya observes (❻) the effect of the mitigation actions and adapts (❼) future prediction and mitigation decisions based on the observations from production.

Building Narya to work for a production cloud requires both algorithm designs and systems support. We first describe the core prediction and mitigation algorithms in Section 4 and Section 6, respectively. Section 7 describes the Narya systems design and implementation.

3.2 Key Optimization Metric

Narya's objective is to reduce and minimize the overall customer impact caused by node failures on the fleet. Defining a good cost metric for customer impact is critical for the Narya's decision engine to optimize that metric. In Azure, we focus on the Annual Interruption Rate (**AIR**) defined as:

$$AIR = \frac{\text{VM interruption count in } T}{\text{Total VM lifetime in } T} \times 365 \text{ days} \times 100 \text{ VMs}$$

T is any given measured interval duration in days. VM interruption in this paper mainly refers to reboots or loss of heartbeats. Internally, we also measure performance drop with a sub-metric we call *AIR-blips*.

We optimize this metric instead of the traditional availability metric for a few of reasons. First, long-duration incidents are now rare in Azure. VM interruptions become more common that require addressing. Second, short VM interruptions can significantly disrupt user experiences, e.g., for gaming-type applications. Third, for VMs that run applications like databases, even if the VM only experiences a short interruption, the applications take time to recover, which translates into a longer user-perceived interruption. Fourth, based on communications with customers, customers can be more annoyed if their VMs get frequently interrupted when compared to a single longer-time interruption.

3.3 Challenges

We need to address several design challenges. First, failure mitigation has to act with incomplete information since the underlying root cause is not known. For Narya, this challenge is even more pressing since the failure has not occurred yet.

Second, due to the massive scale of a cloud system, there are many factors to consider in the decision logic. A decision may work well for some nodes but not others. If not careful, some corner cases can mislead or bias the decision logic. Narya must be robust enough while still being flexible.

Third, our experience suggests that when incorporating failure prediction into a production cloud system, false positives are unavoidable due to the complex system environment, large number of noisy signals, unexpected customer workloads, etc. If the system blindly trusts the failure prediction results and reacts, it could cause unnecessary disruptions. When consuming the prediction results, the mitigation mechanisms should take this into account and operate in a way that minimizes the impact due to unavoidable false positives.

Lastly, failure mitigation is a mission critical procedure. If not designed well, a decision engine may do more harm than good. Ensuring safety should be a top priority for Narya.

4 Predicting Node Failures

The first step in Narya is to predict a host failure before it occurs. In this Section, we describe two prediction methods Narya uses: (1) static threshold rules written by domain experts; (2) machine learning model-based prediction.

4.1 Input Signals

Narya consumes telemetry signals from the entire stack of the host environment to make informed prediction. For hardware and firmware, the monitoring agents collect low-level logs from disk SMART attributes, memory (*e.g.*, uncorrectable errors), CPU (*e.g.*, machine check error), motherboard (*e.g.*, bus error), *etc.* A higher-level source of signals comes from device drivers, *e.g.*, timeout events. Repetition of such events is often an indicator of an imminent failure. Faults in individual component do not necessarily cause customer impact. Some could be transient that would go away after retries. Others may be tolerated by redundancy. Narya further consumes critical OS events and aggregate application performance counters.

Another important source of signals used by the predictor are results from the control-plane operations. For example, repetitive VM creation operation errors could indicate serious host issues even if the host still appears to be running. Such signals help reduce the observability gap [16].

4.2 Rule-based Prediction

Rule-based prediction leverages domain knowledge from hardware, firmware and software experts. We analyze the common failure patterns and the available telemetry signals to predict failures that have significant customer impact. For example, in some cases, CPU Internal Error (IERR) is a good indicator that the node will fail *again* soon; a prediction rule could be marking the node if IERR occurs twice within 30 days. Rules are typically written as Json files, Python scripts or sometimes C++. Since rules are manually written, they are simple and easy to understand. The prediction rules are deployed directly in the host and can be executed fast.

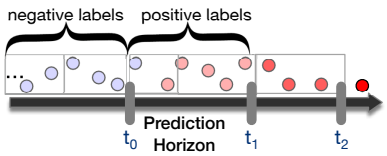


Figure 2: Prediction horizon and label timeline. t_1: host failure time. t_2: component permanent failure time.

Rule-based prediction works best for definitive signals that indicate some severe issue with high confidence. An example is the `AvailableSpare` signal in NVMe device health log. When it drops below a certain threshold, we know the device is almost at the end of its life.

Since many failure signals are not definitive, rule-based prediction cannot cover a wide range of imminent failures. In addition, the prediction may come late and do not provide enough lead time for the mitigation engine. The number of prediction rules also keeps growing, which becomes a burden to manage and tune. We have a total of 51 rules in use.

4.3 Learning-based Prediction

To address the limitation of rule-based prediction, Narya employs an additional learning-based predictor, which analyzes more signals and patterns during a larger time window. It can predict many complex host failures. It also can predict earlier, thus leaving longer time for the mitigation engine to react.

Prior work predict disk failures [14, 27, 38] and node faults [25] with supervised learning. Our learning-based prediction aligns with prior solutions. A main difference is that we focus on overall host health and failures that result in customer impact, instead of failures of individual components. Because of this, Narya analyzes more diverse signals across layers such as control-plane operation signals.

Prediction Horizon and Label. Deciding the labels to use for learning is crucial for Narya. In prior work that predicts hardware failures for alerting, the positive labels are signals close to when the hardware is completely broken. For Narya, the host view of a failure is different from individual components' view. The host failures could be unresponsive host, VM creation failure, host OS crash, *etc*. In our observation, they happen much earlier than the permanent failure of a component (e.g., disk unusable). As Figure 2 shows, if we assign positive labels from time t_2 (permanent component failure), it can yield late prediction which comes after host failure at time t_1. The consequence is that the prediction does not give enough time for Narya to take proper mitigation actions.

Additionally, certain faults might not be a problem to the source component but could be problematic from host's view. For example, a series of memory correctable errors might seem fine for an ECC DRAM because they are corrected. But the host may already suffer slowness and impact VMs.

To get accurate and useful prediction result, we only use host failures that result in customer impact and are later confirmed to be caused by some hardware component faults during diagnosis. For a given host failure, if it occurs at time t, we assign positive (failure) labels for signals from $t-1$ to $t-n$, where n is the prediction horizon and using an hour unit. We assign negative (normal) labels for signals from $t-(n+1)$, ... We also sample negative labels from healthy nodes.

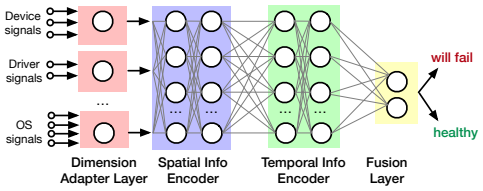


Figure 3: Deep learning model structure.

In production, our prediction horizon is set to 7 days. We made the choice based on how discriminative the feature will be given different horizons. Specifically, we looked at the feature distribution of failed nodes and measured the same distribution of healthy nodes. We then measure the similarity between the two distribution groups. Beyond 7 days, we could not observe a significant difference.

Machine Learning Model. With the signals, labels, and host metadata, Narya trains a binary classifier. The predictor outputs the failure probability of a host (we use 0.5 as the cutoff).

To train the classifier, we use the gradient boosted tree model [18] commonly used in supervised learning, which combines decisions from a sequence of simple decision trees with a model ensembling technique called gradient boosting [10]. This simple model works fine for our scenario in terms of its predictive power, but we have to carefully craft aggregated features from the signals. We engineer 2k+ features from 100+ time series data. Those engineered features are combined with other categorical features to build the learning-based model feature set.

We further explore reducing the feature engineering efforts by directly learning the features with an attention-based deep learning model [20, 35]. At a high level, we aim to learn both spatial features and temporal features. Spatial features compare one component to its neighbors. For example, one host often has multiple disks configured under RAID 0, thus they are expected to perform similarly. If one disk performs worse than its neighbors, it could indicate imminent host failures. Attention-based deep models are designed to capture such patterns so that more weights (attention) are assigned to anomalous neighbors. The temporal features characterize changes in components over time.

Figure 3 shows the structure of this model. First, we use a dimension adapter layer to unify the dimension of signals from different sources. Next, we employ a spatial information encoder based on self-attention. It calculates weights of a component's neighbors. The weighted sum of the neighbors' feature vector represent its spatial information. Then, we use the temporal information encoder, which consists of positional encoding, self-attention, and location-based attention layers. Finally, we employ a fusion layer to do binary classification. We omit the attention implementation details as they are based on an existing technique proposed by Lee *et al.* [20].

	Action	Description
Primitive	Avoid	Deprioritize new VM alloc. on this node
	Unallocatable (UA)	Block new VM allocations on this node
	Live Migration (LM)	Migrate VMs to other nodes on the fly
	Service Healing (SH)	Discon. VMs, move them to healthy nodes
	Soft Reboot (SR)	Reload host OS kernel, VM states preserved
	Human Investigate (HI)	Shut down node and send it to diagnostics
Composite	UA-LM-HI	Block alloc., attempt LM and HI after T
	UA-SR	Block alloc., attempt soft reboot
	UA-LM-RH	Block alloc., LM and unblock after T
	Avoid-RH	Avoid alloc. to this host if possible

Table 2: Primitive and composite mitigation actions for Narya. Composite actions are sorted by decreasing priority.

Overall, this model achieve 5–10% improvement compared to the decision tree model with hand-crafted features.

5 Mitigation Actions

When a host is predicted to fail, Narya chooses among several possible actions. Table 2 lists the main *primitive actions* in Azure. Mitigating a failure often requires multiple primitive actions. An aggressive goal for Narya is to explore the actions arbitrarily and figure out the optimal combination. But this could potentially bring significant customer impact. Instead, Narya mitigation engine focuses on exploring pre-defined *composite actions* (Table 2). This set of composite actions is constantly enriched with new combination and by modifying parameters (e.g., unallocatable duration).

Live Migration moves a running VM from one host to another with minimum disruptions. The migration process involves transfer of the VM's memory, processor and virtual device state [7]. The LM engine iteratively copies the VM's memory pages while maintaining a dirty page set for the VM on the source host. Based on the dirty page rate, network bandwidth, the engine determines the maximum iterations to stop the VM. After the VM is stopped, the LM engine synchronizes the dirty state with the target and resumes the VM on the target host. Note that not all VMs are eligible for LM and LM could fail for various reasons.

VM Preserving Soft Reboot preserves the VM state across a reboot of the host OS. At a high level, the host OS kernel is reloaded into memory, the VM memory and device state are persisted to the newly loaded kernel. The host reboots into the loaded kernel while preserving the persisted state. Once the reloaded kernel starts, the persisted state is restored and the rest of the state in the prior kernel are discarded. The restored VM experiences a brief pause similar to the live migration.

Service Healing is used to restore the service availability of unhealthy or faulted VMs. Live Migration can move running VMs transparently, but it could fail or cannot be applied due to certain constraints such as network boundary. Service healing works for more general scenarios. The VMs will be isolated by powering down or disconnecting from network. The controller generates new assignment of the VM to healthy nodes. During the process, there is some interruption.

Mark Unallocatable blocks allocation of new VMs to a host for some time T (default 7 days). Composite actions typically start with marking a suspected host unallocatable. In *UA-LM-HI*, after marking host unallocatable, the controller attempts to live migrate the VMs on this host to other hosts. After all VMs have been migrated or destroyed by customers or this host fails, the host will be sent to diagnostics. If at the end of the unallocatable period T some VMs are still running (*e.g.*, because they are not eligible for LM) we service heal them before pushing the host to diagnostics. *UA-LM-RH* is a variant of *UA-LM-HI* where we unblock allocation (reset node health) at the end of T. In *UA-SR*, the controller blocks the allocation and then try the kernel soft reboot action. If the soft reboot succeeds, the controller unblocks allocation. Otherwise, we use a fallback strategy, typically *LM-HI*.

Avoid informs the allocator to try to avoid adding new VMs on this host. Blocking allocation has a strong impact on capacity since the host is not eligible for getting new VMs. Thus, the number of hosts that can be marked unallocatable at the same time is limited. *Avoid* action provides a weaker constraint. The behavior on host failure is still to send it to diagnostics. At the end of T, we reset the node availability.

NoOp is a special action for predicted failure, in which the controller does not take any action. This is the baseline to measure the benefits of prediction and taking actions.

6 Decision Logic for Adaptive Mitigation

With different prediction rules/models as well as different mitigation actions, relying on static assignment based on domain knowledge to map each prediction to an action can soon get intractable and ineffective. This motivates the design of Narya decision engine for adaptive mitigation.

6.1 Online Experimentation with A/B testing

One straightforward way for choosing mitigation action is to *estimate* offline the impact for each possible action for a predicted failure. In our experience, given the complexity of cloud systems, it is extremely hard to estimate the impact of actions and know which one performs best without trying them in production. Based on this insight, Narya takes an online experimentation approach to evaluate different mitigation actions by testing them at scale.

A/B testing, also called online experiments, is widely used [19] in front-end designs to test the effect of UI features. Narya adopts the A/B testing methodology and adapts it for discovering good mitigation actions. In classic A/B testing, one experiment is about one UI feature and each unit is a user. For Narya, one experiment is about the mitigation of one failure prediction (*e.g.*, CPU IERR, slow memory access latency), and each unit is a failure mitigation request about a node marked by the corresponding prediction rule/model.

The workflow of the A/B testing in Narya is as follows: (1) each predicted node is assigned to different action groups with

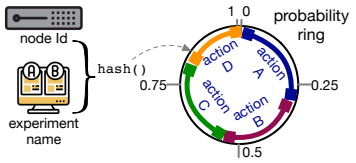


Figure 4: Hash node ID and experiment name for sticky assignment.

equal probability (2) after taking each action, we measure the customer impact within an observation window; (3) we use hypothesis testing to test if an action yields significantly less customer impact than others; (4) once statistical significance is reached, we consider this least-impacting action optimal and apply it for all nodes; (5) we keep monitoring the customer impact per node for the used action; (6) if customer impact significantly increases, we run a new A/B testing experiment to validate that the action is still optimal.

Cost. The cost ($= -1 \times reward$) models the customer impact. It should balance the trade-off between key metrics of our system. The pros and cons of each action should be modeled into the cost to correctly optimize for the mitigation action. We use the number of VM interruptions in the node during an observation window and the VM interruptions in nodes to which we migrated VMs in live migration or service healing.

An additional constraint we need to consider is capacity. As capacity does not directly impact customers and is not visible at the node level, it cannot be easily added into the cost. We currently incorporate the constraint by limiting the number of nodes that can be marked unallocatable for the same rule in the same cluster at the same time. With this, capacity indirectly impacts the cost of marking nodes unallocatable.

Assignment Strategy. A crucial point for A/B testing is to decide for each node marked by the failure predictor, in which experiment group should it go to. In classic A/B testing, each experiment unit is assigned randomly, based on the assumption that the units are independent and identically distributed (i.i.d). For Narya, we make several changes.

The same node can be marked by the same prediction rule multiple times during an A/B experiment. In this case, if Narya assigns it different mitigation actions, e.g., assigns node X in action A group at time t_1 and then to action B at time t_2, the i.i.d assumption can be violated. This is because the underlying node condition at t_1 and t_2 could be highly correlated, especially for hardware issues. Then the observations from the treatment and control group are correlated.

To address this issue, we introduce **sticky assignment**: for each node, the group is determined through the hash of the node Id and experiment name (Figure 4); then, if a node is assigned to action A for an experiment, it will always take action A for subsequent requests.

Classic A/B experiments are typically done sequentially. In our case, sequential experimentation takes too long; so Narya allows different experiments to take place concurrently. While most experiments are independent, some experiments could have prediction rules that are correlated. In this case, it

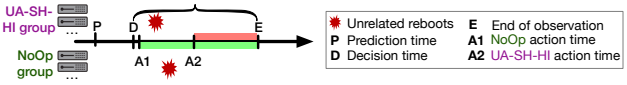


Figure 5: Using decision time as the start of observation window.

is important to test all possible action combinations to analyze their compound effect later. For example, with experiment X testing actions `{a,b}` and a correlated experiment Y testing actions `{c,d}`, we need to have observations that take each of the four scenarios: `(a,c)`, `(a,d)`,`(b,c)`, `(b,d)`.

Action Overriding. Since many fault handling policies, including our A/B experiments, can take place concurrently, a host can potentially be marked by several prediction rules. As a result, the host might need to follow different composite actions at the same time. A common reason for this is the incomplete information factor (Section 3.3). To handle this situation, Narya uses a specific override logic based on the priority from the order in Table 2. When we try to assign a node with a lower-priority action than its current one, we skip it. In case of equal priority, Narya honors the older actions. The rationale is that later prediction can often be a side effect of the earlier one. Since we often do AB testing between actions with different priorities, it is critical to also observe cases where the action was skipped or later overridden.

Effect Observation and Attribution. Depending on the complexity of the actions, some need longer time to get triggered. In the time between the decision and the start of the action, unrelated VM interruptions can happen. However, to fairly compare action, we should still count the cost in this time gap because, for instantaneous actions, it would be impossible to differentiate the costs caused by the action from the unrelated ones. Like for overrides, we monitor decision instead of action, hence we use the decision time instead of the action time as the start of the observation window. Figure 5 shows an example where if we used the actual action time we would ignore some unrelated reboots for one action and not the other, while they happen in both groups.

Hypothesis Testing. After collecting the cost metrics for each action, the decision engine performs hypothesis testing to decide whether an action is optimal and the experiment can be stopped. Since our cost function is complex and depends on external variables, we simplify the hypothesis testing by assuming that the number of VM reboots per node is i.i.d and follows a normal distribution. Since different actions can highly change the VM reboots per node, we will not assume equal variance for the different actions. Under these assumptions, we use Welch's t-test [36] when testing for two actions and Welch ANOVA test for 3 and more. In the latter, we use post-hoc analysis to remove all statistically worse actions until one action remains.

6.2 Bandit Modeling

One drawback of A/B testing is its static group assignment. Before statistical significance, we do not leverage the estimated difference between the groups to minimize our cost,

and once an experiment reaches statistical significance, we will almost always use the discovered optimal action. This is essentially the classic *exploration-exploitation* dilemma. Naturally, we explore modeling the adaptive mitigation as a *Multi-Armed Bandits* problem [32, 33], where we aim to minimize the customer impact over time by ensuring a balance between exploring potential better actions and exploiting the discovered best action. At training time, we observe tuples `(node, rule, chosen action, cost)` to estimate the probability to choose each action, while at serving time, we match a request tuple `(node, rule)`, to the learnt action.

Actions. The output of the bandit model is the composite action we want to attempt. The available actions are typically defined per experiment based on offline analyses of the prediction signals characteristics: false positive ratio, time to failure/impact and actions feasibility (Section 5).

Exploration Algorithm. To minimize customer impact over time, we face the classical exploration-exploitation trade-off. We need to explore different actions to see which one minimizes the customer impact but at the same time we want to use the action with minimum estimated cost as much as possible. In other words, we need to balance between short-term and long-term benefits. We experimented with multiple different exploration models including Epsilon Greedy and UCB, and decided to use *Thompson Sampling* model since it provides more explainability and continuous probability changes. In Thompson Sampling, we model the reward as a function of actions and a model parameter and choose the action according to the probability that it maximizes expected rewards. This Bayesian approach updates the prior using observations of actions taken and chooses each action with probability equal to the chance that it minimizes the expected cost:

$$P(a*) = \int I\big(E(c|a*,\theta) = min_a E(c|a*,\theta)\big) P(\theta|obs) d\theta$$

where $P(a*)$ is the probability to choose action $a*$, θ is a hidden parameter, c is the cost and obs are the past observation as list of tuples (a_i, c_i). Our technical report [**?**] describes in more detail the Thompson Sampling algorithm in Narya.

6.3 Extension to Bandits

Compared to traditional Bandits, our system faces several challenges. In addition to the effect observation solution described in Section 6.1, we made 4 main adaptations as follows.

Accommodate Temporal Changes. Since our system can change in time, older observations will gradually become less and less relevant. To account for this factor, we use an exponentially decaying weight for observations to focus on recent data. We will apply a multiplying weight to past observation in the format of $decay = \sigma^{T-T_{obs}}$, where σ is the decaying factor, T is the current time and T_{obs} the time of the observation. We set σ by default to 0.99 based on simulation-based experiments and so that the weight would be close to 0 after 3 months which is our typical retention policy. In the case of Thompson Sampling with Gamma Prior, the distribution to sample from becomes:

$$P(\theta|a, obs) \sim \Gamma\left(1 + \sum_{i,a_i=a} c_i \sigma^{T-T_i}, 1 + \sum_{i,a_i=a} \sigma^{T-T_i}\right)$$

Delayed Reward Collection. A key challenge in our settings is the potential long time between the action taken and its impact. This forces us to observe for at least 10 days and up to 30 days the impact of choosing each action. This observation window highly depends on the duration of the action and its effect: UA-LM-HI for 7 days would require around 10 days while Avoid-RH for 15 days would require a full 30 days to observe potential failures following the health reset. The drawback of a long observation window is the delay for the reward to be integrated into the model. Thus, wrong estimation could be used for a while before the observed cost can readjust the probabilities. One way to counteract this effect is to observe the reward as it comes. But we can suffer from the opposite effect of getting biased by reboots close to the decision time. Our experience suggests that we need to wait for the full initial observation window and then can collect partial rewards incrementally.

Bandit stickiness. Since the probability to choose each action over time changes, we cannot rely on a hash function like in A/B testing to ensure a node is always assigned to the same action. We define the bandit stickiness for time T as reusing the previously chosen composite action if the node has an available decision for the same rule within the T time window.

Deal with Unexpected Spikes. Another potential issue in our system is the unexpected spike of VM interruption events that could affect one action group more than the other. One approach would be to perform an outlier removal step before using such observation, but in that case it could also filter out spikes inherent to a specific action, which should be integrated into our learned model. We address the issue with the safe guards mechanism described below.

6.4 Safe Guards

Safety is a top priority in Narya mitigation decision logic. We take several measures to ensure safety. In addition to action overriding (Section 6.1), we also apply *safety constraints*—domain-specific restrictions to prohibit certain actions in some failure scenarios. Narya decision engine also requires a minimal number of observations before following the recommendation from the Bandit. The Bandit model will output a *premature* flag for insufficient observations, in which case we would fall back to default action probabilities similarly to A/B testing. This also helps dilute the potential effect of spikes in a larger observation set.

Moreover, we support configuration of minimum and maximum constraints for each action probability. The maximum constraint limits the possible reactions to high cost, while the minimum constraint guarantees some exploration for actions that could seem irrelevant at a specific time. In practice, any

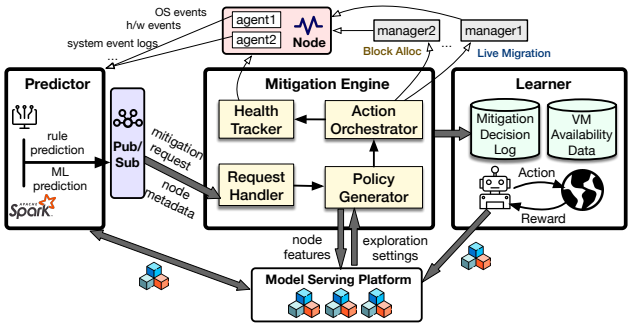


Figure 6: Narya system architecture, which consists of the predictor, mitigation engine and learner. The ML models for the prediction and mitigation are stored in a model serving platform [8].

experiment for which we expect a potential change in the system should keep a minimum probability for each of the allowed action so that it will continue observing the effect of such action. Experimentally, We found that using 10% exploration when nodes flagged per day is less than 100 and 5% when it is higher yielded the best results.

7 Narya System Design and Implementation

In this Section, we describe the system support for Narya. Figure 6 shows the system architecture. Narya is deployed in each data center region of Azure compute. The Narya system must be able to process the massive signals and requests from the entire fleet with low latency and reliability.

7.1 Failure Predictor

Azure deploys various agents in each node to monitor the health of the host environment. The Narya predictor ingests health signals from these monitoring agents and runs rule-based prediction and ML-based prediction (Section 4).

Rule-based prediction has low cost and high priority. Thus its prediction logic is executed directly in the host. The ML-based prediction inspects much more signals such as performance counters and runs more complex prediction logic. Thus, the ML predictor is implemented as a centralized service. It collects raw signals from monitoring agents using micro-batches (small groups) and incrementally processes them. Open source technologies are used for ML modelling. LightGBM is used for decision tree model and PyTorch for deep learning model. The ML inference tasks run as an hourly Spark job, which reads the most recent signals and the trained ML model to compute failure probability for each host.

Pub/Sub Service. A *mitigation request* is created if a node is predicted to fail with high probability. The predictor publishes the request along with metadata information about the host (e.g., hardware generations, OS version) to a central pub-/sub service, which we implement on top of Kafka [1]. We choose Kafka because it allows scalable, low-latency, and real-time streaming processing to deliver the mitigation requests quickly. Also, our computation pattern involves many

```
"HW_Triage": {
  "Type": "Selection",
  "ChildSelector": [{
    "ShouldSelect": "C#|Request.FaultedHwHealthGrade == 100",
    "Child": "HW_Try_HI"
  }, {
    "ShouldSelect": "C#|Request.FaultedHwHealthGrade == 75",
    "Child": "HW_Try_Unallocatable"
  }]
}
"HW_Unallocatable_WithRecovery": {
  "Type": "Action",
  "Actions": {
    "MarkNodeUnallocatableAction_WithRecovery": {
      "Action": "MarkNodeUnallocatableAction",
      "Input": {...}
    }}
}
```

Figure 7: Example of mitigation policy tree nodes.

data producers for a small number of consumers, which is a main scenario Kafka is designed for.

7.2 Mitigation Engine

The mitigation engine is a core component of Narya. Internally, it is composed of four major microservices. These microservices communicate with each other and other services in Azure using REST APIs.

Create Mitigation Job. The *Request Handler* microservice consumes mitigation requests from the Pub/Sub service. Upon receiving a mitigation request, it creates a mitigation job with a job Id. This job Id is used by other micro-services to track the mitigation and query its progress.

Instantiate Mitigation Policy. For a new mitigation job, the *Policy Generator* creates a *mitigation policy*, which maps the information from the request to the action to take. It is represented as a decision tree. There are two types of tree nodes: a `Selection` node, which chooses the tree node to visit next based on some C# predicate; an `Action` node, which executes a user-defined C# function. The decision tree structure allows us to easily specify the decision logic. For example, we can decide mitigation actions based on failure types (software or hardware), fault codes (e.g., `Req.FaultCode == 0x123`), cluster types (storage or compute), hardware generations (e.g., `HostNode.Gen != "ABC"`), etc. Figure 7 shows an example.

The policy is derived from a template (Json configuration file). For A/B testing, the action distribution is specified in the template. In the `Action` tree node matching a mitigation request, the node's C# function samples one mitigation action from the distribution. For Bandit, the `Action` tree node dynamically generates the distribution based on contextual information. In particular, it calls a ML model serving platform, Resource Central [8], with relevant features such as the fault code, VM count, etc. The platform returns an exploration setting—a probability distribution over mitigation actions.

The policy generator then applies safety constraints on the retrieved exploration setting to obtain an adjusted action probability distribution. Additionally, the mitigation policy allows imposing *rate limit* for a tree node to avoid excessive mitigation that could cause capacity issue or cascading failures.

Walk Policy Tree. The policy generator further traverses the

policy tree in DFS order and creates an *action plan*. During this process, the generator performs many steps such as checking predicates, checking rate limit condition etc. If a node is entered, the generator first checks if we need to apply *sticky* mitigation action (Section 6.1) and if there were decisions made within certain period of time for the same experiment and tree node. In that case, the last mitigation action is retrieved from a distributed storage service.

Otherwise, one action is sampled based on the probability distribution from the config if in A/B testing mode and from Resource Central if in bandit mode. If there is insufficient data learned in bandit mode, a specific flag is returned to indicate the Bandit model is pre-mature. The generator then falls back to use the action probabilities from A/B testing mode. This allows us to bootstrap bandit learning from A/B testing safely, especially considering the delayed cost in the feedback loop. We follow the same fallback strategy if there is any error in calling the model serving platform.

Carry out Action Plan. The *Action Orchestrator* microservice is responsible for carrying out the action plan from the policy tree walk session. This step involves making API calls to the corresponding compute managers since different actions may be implemented by different managers. The orchestrator executes actions asynchronously to avoid blocking.

Log Actions. Logging in general is very important for data analysis, Bandit training, and counterfactual evaluation of different mitigation policies. The logging format for Bandit learning is special since it requires not only recording the chosen action but also the associated probability. In particular, the mitigation engine will log the action timestamp, experiment name, model type, model name, model version, action distributions, chosen action, chosen action parameters, etc.

Track Node Health. The *Health Tracker* tracks node and VM health information during the mitigation process. For example, while rebooting a node, if we get a new signal (e.g., a WindowsEvent) that it is a hardware issue, then we can HI the node early instead of waiting for reboot to fail/timeout.

7.3 Learner

Learner is a centralized component in Narya. It learns the effect of mitigation action across different data center regions. Compared to a regional learner design, a global learner has the advantage of observing more data points and hence more confidence in the cost estimation. Additionally, a mitigation effect change in certain region due to software/firmware updates could be quickly learned and applied to other regions rolling out the same updates.

The learner runs two main jobs: cost collection and Bandit model training. The cost collection job retrieves the mitigation engine's decisions from the logs. This information is then correlated with the VM availability measurements and other important information (LM status, VM workload, etc.) to determine the cost of the mitigation action for training. The

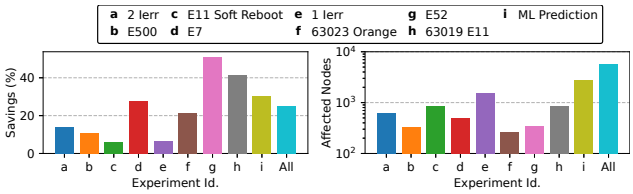


Figure 8: AIR improvement of all A/B testing and Bandit experiments in March 2020, breakdown per experiment.

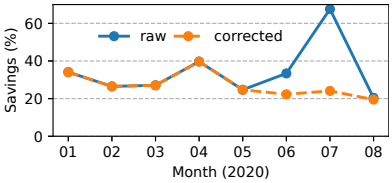


Figure 9: AIR improvement per month.

Bandit model training runs on a Spark cluster. The output model of the learner is a categorical distribution, which the model server can easily draw samples from.

8 Evaluation

Narya has been running in production since June 2019 to prevent VM interruptions in Azure compute platform. Our evaluation answers several questions: (1) how effective is Narya in averting VM interruptions? (2) how accurate and timely is the failure prediction? (3) how does Bandit model compare with A/B testing?

8.1 VM Interruption Savings

The main metric we use to evaluate the effectiveness of Narya is the VM Annual Interruption Rate (AIR) (Section 3.2). We measure the delta between the AIR using the old static assignment and the AIR under new mitigation decisions from Narya. We specifically compute three metrics: the estimated daily AIR savings, the oracle daily AIR savings (savings if we already knew what was the best action), the regret (how much additional AIR we could have saved). The estimated daily AIR savings ($\hat{S}$) is obtained by comparing the impact of each tested action to the impact of the original action projected on the whole fleet. The oracle daily AIR savings (S^*) is estimated by mapping the best performing action to the whole population compared to the original action on the whole fleet. Our technical report [**?**] shows the formulas to calculate $\hat{S}$ and S^*. The regret is the expected difference between the reward sum associated with an optimal strategy and the sum of the collected rewards. We consider $R = S^* - \hat{S}$ to be our AIR regret, meaning how much additional AIR we could have saved if we knew the best action all along.

Due to the confidentiality nature of AIR, we report $\hat{S}, S^*, R$ as relative percentages compared to the overall AIR contributed by all of our target failure types (host failures caused by various hardware problems). For the month of March 2020, $\hat{S}$ **is a 26.2% improvement**, i.e., Narya successfully

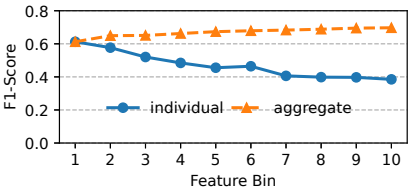


Figure 10: Contrib. of different features.

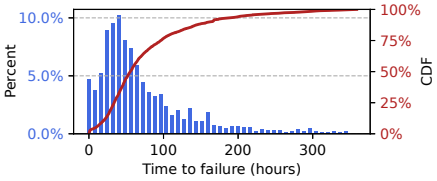


Figure 11: ML prediction time to failure.

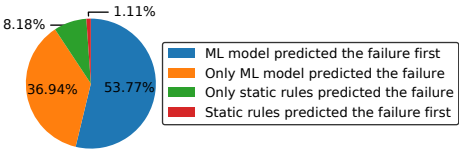


Figure 12: Prediction timeliness.

decreases AIR by about 26.2% compared to the static strategy. We also provide the per-experiment improvements in Figure 8. This shows, for each AB or bandit experiment what percentage of VM interruptions were prevented compared to using the original strategy. 63019E11 constitutes the largest savings since the AB testing experiment was already using the best action. For the same period, S^* is 35.4%, meaning the best AIR reduction percentage we could possibly achieve (if we use the optimal action); R **is 9.2%**.

Although a 26% might not seem big, given Azure's scale, it represents a large number of VM reboots, each highly impacting to customers. The ad-hoc mitigation strategy has already been tuned for years, and the overall availability of Azure is already high. Therefore, the comparison baseline is not low. Also note that the oracle saving 35% is only with respect to the online experiments we have run. With new prediction rules for more failure types and new actions, Narya can potentially yield further improvement.

8.2 Savings Trend Over Time

Figure 9 shows the AIR improvements over time. Overall, the savings fluctuate between 20% to 40%. July saw a sudden jump. This is because one major firmware fix occurred in the June-July time period. One rule started marking much more nodes, including nodes considered false positives (not likely to fail soon). This largely increased $\hat{S}$ as the old policy was very sensitive to false positives. However, our anomaly detection caught this issue. We probably would have fixed the policy if we were using it. We report in Figure 9 the corrected savings assuming that fix. In addition, this firmware deployment fixed a driver issue for which our mitigation was reducing much AIR by predicting failures in advance. As a result, our savings decreased in July and August.

8.3 Accuracy and Timeliness of Prediction

We first measure the precision and recall of the Narya failure predictor. There are multiple rules or models to predict different types of host failures. For a prediction rule/model r, we define the prediction precision as $\frac{F+D}{N}$, where N is the total number of hosts marked by r; F is number of hosts that fail and are diagnosed to be indeed caused by the suspected fault; D is number of hosts that Narya successfully mitigate and the suspected fault is later confirmed. The recall is defined as $\frac{F+D}{M}$, where M denotes the number of host failures that are diagnosed to be caused by fault type that r represents.

The overall precision is 79.49%, while the overall recall is 50.7%. While the false positive rate (20.51%) is not small, we note that failure prediction in a large-scale, complex, and frequently changing cloud like Azure is an extremely challenging problem. Narya is designed with the expectation that false prediction is unavoidable and employs several measures such as low impact actions, safety constraints, longer observation window to minimize the impact of false prediction.

We further evaluate the contribution of different signals (features) to the prediction accuracy. We calculate the feature importance using the SHAP method [28], sort features by their importance, and group them into 10 bins. We then evaluate the precision and recall (F1-score) using individual bins or aggregate bins (features from bin #1 to #N). Figure 10 shows the result. We can see that some features are more important than others. The first bin in particular contributes significantly. Examples of features in the first bin include read error rate, flush count, AvailableSpare, HostReadCommands, etc.

Besides precision and recall, for Narya, It is crucial to consider the prediction lead time (or time to failure) defined as the duration between the prediction time and the failure time. A larger lead time gives Narya more time to take preventive actions. Figure 11 shows the CDF of the ML prediction time to failure with a median lead time of 2.44 days. Figure 12 compares the timeliness between the ML-based prediction and rule-based prediction: ML-based prediction provides significant advantage in timeliness.

We measure the quantitative benefit of early prediction to live migration success. For nodes predicted to fail, the average successful live migration per node is 5.57. With a smaller lead time, the successful live migration per node gets down to 3.

8.4 Comparing AB Testing and Bandit

Next we compare Narya Bandit and A/B testing in finding the optimal mitigation action. To do so, we compare the used strategy to the other possibilities. For mitigation requests that go through A/B testing, we use off-policy learning of Bandit based on observed A/B data. Similarly we compare the Bandit output to a static A/B policy. We then use counterfactual estimation [32] with Inverse Propensity Score to estimate the cost of running Bandit in place of A/B testing. Over the 2 months period of February and March 2020, using Bandit instead of A/B testing for ongoing experiments could have helped decrease the number of VM interruptions by 14.4%. The breakdown per experiment can be found in Figure 13. Note that A/B testing compared to bandit is safer and allows for more than one cost metric.

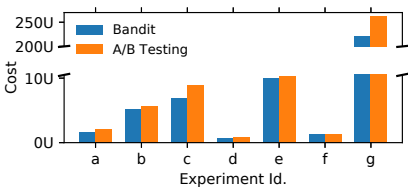


Figure 13: Cost metric (VM interruptions, units anonymized) under Bandit vs. AB testing.

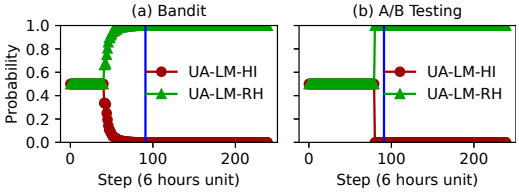


Figure 14: Median probability of choosing each action using Bandit and A/B testing over 1000 re-sampling simulations based on production data.

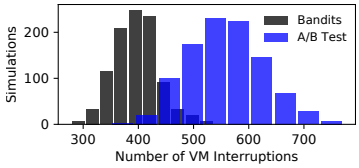


Figure 15: Distribution of reboots using Bandits or A/B testing in 1000 simulations based on production data.

Min	Median	Max	No Convergence	Ongoing
12 days	29.5 days	140 days	2 experiments	3 experiments

Table 3: AB testing experiments convergence time.

8.5 Convergence to Optimal Action

Table 3 shows the convergence time of A/B testing for different experiments. The variance of the convergence time is big, because we need to accumulate enough samples. For different rules, the time it takes to collect the samples differs a lot. Two experiments did not reach convergence at the end of experiments. No convergence usually indicates that there is no significant difference among the experimented actions. For Bandit, we compare its behavior with A/B testing through simulation with production data (details in report [**?**]). Bandit can achieve a much faster convergence to the best action. Figure 14 shows the result. Bandit converged in around 50 steps, while AB testing would converge in 125 steps. Faster convergence also yields more AIR savings. As Figure 15 shows, Bandits yields much fewer reboots than AB testing.

8.6 Case Studies

Blobcache error is a symptom that can caused by hardware issues or some recoverable faults. The original mitigation policy was *UA-LM-HI*. We wanted to test if we could avoid the impact of service healing at the end of the unallocatable period and try to reset node health (unblock allocation) instead. We used an AB testing experiment to compare the VM reboots per node when using *UA-LM-HI* and *UA-LM-RH*. The experiment started on 03/10 and statistical significance was reached on 03/25 and observed on 04/02. We were able to save 25.2% AIR compared to the old policy during this period. Once, we adopted the new policy (config change to make it 100%) and deployed it to production, it saved 50.3% of AIR associated with this type of failure.

E11 is a Windows event indicating a disk controller error. When this event occurs, it generally means that the hard disk is experiencing some issues most likely indicating an imminent failure. Using offline correlation analysis on non-empty nodes with no prediction from other rules/models, we found that 85% failed in the following 7 days, with a majority in the first few hours. Although the lead time was small, it should have sufficed to migrate some VMs in each of these nodes.

We started our first AB testing experiment on 2019-07-25 to try our *UA-LM-HI* policy over *NoOp*. To our surprise, we did not see the significant impact we imagined. Upon analysis, it appeared that few of the VMs could live migrate, mostly because the node failed too quickly or the disk state was already too bad to succeed in live migrating the VMs. However, after a few month of AB testing, the difference in reboots revealed to be significant mostly through short lived VMs getting stopped and repeated failure being avoided by the unallocatable policy. The experiment was ended on 2019-12-12 with approximate daily AIR savings of 28% for such signature.

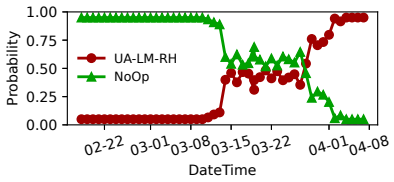


Figure 16: Action probability change using Bandit decisions.

We started another AB testing experiment on 2020-01-11 to test the use of soft reboot to mitigate some of these issues, following a few positive offline test. Contrary to our belief, we observed no significant improvement, even a slightly larger VM reboot per node in the *UA-SR* action. In total, 51 nodes tried the *SR* primitive action, with none of them succeeding because some pre-checks were not met. This shows the importance of AB testing all potential new actions before using them on the whole fleet.

I/O Timeout We started an AB experiment between *UA-LM-RH* and *NoOp* actions on 2020-01-11 for an I/O timeout prediction rule. At first, *NoOp* seemed to be the better option, although not significantly. In late March, the *UA-LM-RH* action started being consistently better and led to us to switch to Bandit. As Figure 16 shows, we can see a clear switch from choosing *NoOp* to choosing *UA-LM-RH* for that time. Even though we are unsure as to what system changes trigger the probability change, our Bandit model picks up the changes and adapt. Using counterfactual estimation, the results show the Bandits adaptation yields savings of 3.7%–13.9% compared to both static policies.

8.7 Reward Collection Schemes

We compare three possible reward collection schemes: (a) delayed reward collection; (b) immediate reward collection; (c) incremental reward collection.

Delayed reward collection (§6.3) is our default scheme. In (b), we associate VM interruption costs with an action on the

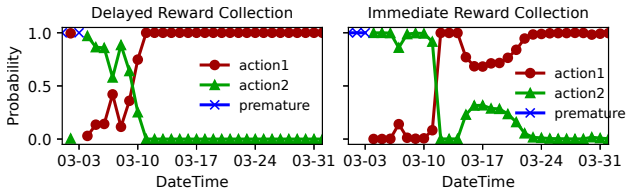


Figure 17: Probabilities in taking two actions using delayed versus immediate reward collection. Action1 is optimal.

day where the action is completed, and we update the Bandit model. In (c), we update and associate the cost daily.

Across different A/B testing experiments we run, the three schemes have varied effectiveness. But overall delayed reward collection outperforms immediate scheme. In particular, (b) yields an average 7.12% AIR improvement, while (a) yields an average 8.50% AIR improvement. This is because immediate reward collection can be easily affected by noises and does not account for action impact that takes a while to manifest itself. Figure 17 illustrates the comparison in one experiment.

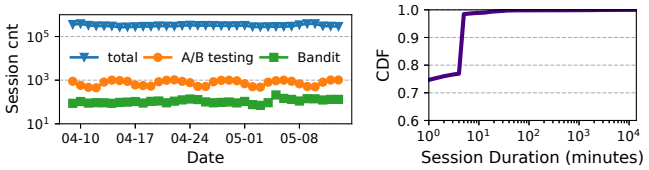
We expected that the incremental scheme (c) would perform slightly better than (a), since it could use more information and would not need to wait for the full observation. Contrary to our expectation, in our experiments, the incremental scheme performs slightly worse than the delayed scheme, with a 8.45% AIR improvement. One reason is that, if the environment is relatively steady, adding partial observation does not provide much new information. Additionally, the collected cost from the partial schema might not be distributed evenly across the observation window. In this case, the incremental scheme would be misled by partial observations. However, the incremental scheme does have the advantage of a faster response to system changes, since it can make decisions based on the latest cost data.

8.8 Safe Guards

The safe guards can influence the system in many ways. First, it allows a constant exploration of all action to enable timely adaptation to system changes. In the I/O time out case studies, the bandit could not have readjusted to use *UA-LM-RH*, hence losing 3.7% of cost. Second, it prevents early convergence to a wrong policy. In the E11 case study, when simulating the bandit without safe guards, we converged (probability > 0.95) to use a single action in 27% of cases, 19% of which was *UA-SR*, the worst action. Third, safe guards decrease the impact of unexpected spikes. In the IO timeout experiment, on 2020-04-25, cascading failures resulted in 106 VM interruptions on a single node for the *NoOp* group. Although this significantly impacted the probability to choose *NoOp*, we still keep exploring that option.

8.9 Scale and Performance

Narya runs in each data center region of Azure. The mitigation engine handles hundreds to thousands of requests daily. The failure predictor processes tens of TBs of health signals per day. Figure 18a shows the number of daily mitigation request



(a) Session count **(b)** Session duration

Figure 18: Mitigation request handling sessions

sessions (including all fault handling), and the number of requests that go through our A/B testing experiments and requests handled by our Bandit model. Figure 18b shows the CDF of the session duration of the mitigation actions.

9 Discussion and Limitations

Lessons. We share some operational issues and summarize the lessons we learned from running Narya in production.

First, given the sheer complexity of Azure cloud infrastructure, it is inevitable that some Narya decision could go wrong. We encountered two kinds of service misbehavior: (1) some prediction rules are outdated and incorrectly mark many nodes in a short period of time; (2) an increase of customer impact that is not incorporated within the cost model. For (1), the issue would impact AB testing and cause Bandit to take time to adjust. Our rate limit mechanism described in Section 7.2 would help. We also designed a separate anomaly detection algorithm to catch such misbehavior so we can pause and refine the offending prediction rules. To overcome (2), we added monitoring of the support tickets filed by customers.

Second, as Narya consumes telemetry signals from the whole stack, Narya may be broken if the updates of host OS, firmware, and hardware involve uncoordinated schema changes. We recently had an issue in which the schema change of a few critical OS signals was not captured by Narya. Our monitoring component caught the issue and we had to patch Narya. Besides schema changes, the data and label quality may also fluctuate due to improvements or regressions of tracing capability introduced by different component teams.

The aforementioned challenges can be addressed if the cross-team collaboration and communication are perfect, which unfortunately is not realistic in large organizations. Through continuous learning from failures, we build multiple channels based on social alignment principles to include relevant component teams, so that the right team can be involved in time to avoid broken contract, adjust prediction rules, etc.

While we try to ensure sufficient communication, we cannot just rely on it. We build a comprehensive monitoring pipeline that detects anomalies at all layers for Narya, from input data to prediction results, mitigation actions, etc. We proactively investigate alerts and follow up on issues caused by external dependencies or fix Narya's own defects. Moreover, we re-train our ML model on a regular basis to accommodate the evolution of telemetry data and label quality.

Third, Narya may output unexpected decisions, which require verifying its correctness and diagnosing the root cause.

In general, diagnosing issues in Narya' Bandit decisions is easy. The exploration model is explainable and solely depends on the total VM reboots observed and the total nodes observed. Any unexpected change in probability can be traced down to the observations that had large customer impact.

Limitations. We describe several limitations of Narya. While Narya can be fully automated, it currently still involves some human intervention to analyze the experiment results and update the system. This is because our cost model for customer impact is incomplete. We believe limited human intervention is key to catch any gaps in customer complaints and improve.

We currently focus on predicting and mitigating hardware or firmware-induced VM failures. We plan to extend Narya to software-induced VM failures. While generic software failure prediction is very challenging due to their frequent changes and complex dependencies, there are potentials for addressing issues like memory leak, repeated crashes, and timeout bugs.

The multi-armed Bandits model we use has the advantage of simplicity and easy explainability of the mitigation decision. However, this model can segment the data. In particular, Narya divides nodes into different experiments based on fault code and node metadata (e.g., h/w generation). But mitigation actions for nodes from different experiments may share the same characteristics, which may not be learned because each model is trained separately. We are exploring the contextual Bandit model [23] to leverage context information like node features to the model input.

10 Related Work

Our work is related to three subareas in system resilience: failure detection, prediction and mitigation. Failure detection has been extensively studied, while failure prediction and mitigation are not as well explored. Narya's major contribution is improving the latter two in the context of a large-scale, production cloud VM infrastructure, and designing an end-to-end preventive mitigation service to achieve failure *avoidance*.

Detecting crash failures reliably and quickly in asynchronous distributed systems is a classic topic [3, 5, 6, 9, 13, 22, 34]. Recent work has discussed the prevalence of gray failures [11, 17, 26] in cloud. Panorama [16] proposes to leverage observability to detect gray failures [17]. Narya focuses on predicting failures ahead of time. Many of the failures we target fall into gray failure category. But our aim is to identify risky hosts *before* they cause customer impact.

Several recent work proposes using machine learning to predict disk failures [14, 27, 38] and node faults [25]. Narya predictor aligns with these solutions' basic approach. But we focus on predicting failures in the complex VM host environment as a whole and only those with customer impact. Additionally, we design the prediction pipeline to closely integrate with the mitigation engine.

The Recovery-Oriented-Computing project [31] advocates the importance of failure mitigation, particularly reboot [4]. Piegon [21] proposes to expose uncertainty of failures to allow better failure reactions for applications. But applications have to manually decide whether to wait or start recovery. IASO [30] is a framework for detecting fail-slow issues and supports mitigating slow issues with multiple options such as process restart or VM shutdown. But it relies on customers to manually configure the mitigation option.

NetPilot [37] aims to automate the failure mitigation in a data center network by determining the suspected network devices and mitigating failures based on estimated impact. Narya differs with NetPilot in several ways. First, Narya targets automating the failure mitigation of a system with heterogeneous components and complex stack. In our setting, estimating the impact of an action offline is challenging and often mismatches with production observations. Narya takes an online exploration and learning approach. Second, the mitigation actions available in NetPilot are few and simple like device restart. Narya needs to consider diverse and complex actions. Third, Narya aims to *avoid* failures whereas NetPilot focuses on mitigating failures that have occurred. Lastly, Narya is deployed in production at large scale.

A/B testing experimentation is a common practice to test the effects of UI features using production data (user requests). The idea is simple, but it often yields surprising power [19]. Thus, leading companies conduct thousands of A/B experiments annually. Narya mitigation engine adopts the A/B testing methodology in a novel way to the failure mitigation scenario with several changes. Narya also adopts multi-armed Bandits reinforcement learning [32]. Our contribution is addressing several unique challenges and the system support that make the approach work in a large-scale, production cloud infrastructure to avert real cloud VM interruptions.

11 Conclusion

We investigate an important topic in fault-tolerant system designs—failure avoidance—in the context of cloud infrastructure. Drawing from our experience in operating a large production cloud system, we propose a novel online experimentation and learning approach to tackle this problem. We present Narya, an end-to-end service consisting of failure prediction and smart mitigation. Narya continually evaluates the optimal action in production using A/B testing and Bandit models. Narya has been running in Azure compute infrastructure for 15 months and yields a 26% improvement in reducing VM interruptions compared to previous static strategy.

Acknowledgments

We would like to thank our shepherd, Haryadi Gunawi, and the anonymous reviewers for their thoughtful and comprehensive comments. We thank all the Azure engineers who partnered with us on building the solution and providing feedback to us. Peng Huang is supported by the National Science Foundation CAREER award CNS-1942794.

References

[1] Apache Kafka: A distributed streaming platform. https://kafka.apache.org.

[2] S. Agrawal and N. Goyal. Thompson sampling for contextual bandits with linear payoffs. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1220–III–1228, Atlanta, GA, USA, 2013.

[3] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS '09, pages 3–3, Monte Verità, Switzerland, 2009.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI '04, pages 31–44, San Francisco, CA, 2004.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[6] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, May 2002.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[8] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, Shanghai, China, 2017.

[9] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, Feb. 2003.

[10] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[11] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Oakland, CA, USA, 2018.

[12] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 8–8, Santa Ana Pueblo, New Mexcio, 2013.

[13] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 175–188, Stevenson, Washington, USA, 2007.

[14] G. Hamerly and C. Elkan. Bayesian approaches to failure prediction for disk drives. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, page 202–209. Morgan Kaufmann Publishers Inc., 2001.

[15] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan. Meaningful availability. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 545–557. USENIX Association, Feb. 2020.

[16] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[17] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[18] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[19] R. Kohavi and S. Thomke. The surprising power of online experiments. *Harvard Business Review*, 95(5):74–82, 2017.

[20] J. Lee, Y. Lee, J. Kim, A. Kosiorek, S. Choi, and Y. W. Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019.

[21] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 427–442, Lombard, IL, Apr. 2013.

[22] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, Cascais, Portugal, Oct. 2011.

[23] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 661–670, Raleigh, North Carolina, USA, 2010.

[24] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20. USENIX, Feburary 2020.

[25] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, M. Chintalapati, and D. Zhang. Predicting node failure in cloud service systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 480–490, Lake Buena Vista, FL, USA, 2018.

[26] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20. USENIX, Feburary 2020.

[27] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi. Making disk failure predictions SMARTer! In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 151–167. USENIX Association, Feb. 2020.

[28] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS '17, page 4768–4777, Long Beach, California, USA, 2017.

[29] J. C. Mogul and J. Wilkes. Nines are not enough: Meaningful metrics for clouds. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 136–141, Bertinoro, Italy, 2019.

[30] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 47–61, Renton, WA, USA, 2019.

[31] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented

computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, EECS Department, University of California, Berkeley, Mar 2002.

[32] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[33] W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 12 1933.

[34] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, The Lake District, United Kingdom, 1998.

[35] X. Wang, L. Yu, K. Ren, G. Tao, W. Zhang, Y. Yu, and J. Wang. Dynamic attention deep model for article recommendation by learning human editors' demonstration. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 2051–2059, Halifax, NS, Canada, 2017.

[36] B. L. Welch. The generalization of 'student's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

[37] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating datacenter network failure mitigation. *SIGCOMM Comput. Commun. Rev.*, 42(4):419–430, Aug. 2012.

[38] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J.-G. Lou, M. Chintalapati, and D. Zhang. Improving service availability of cloud systems by predicting disk error. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 481–493, Boston, MA, USA, 2018.

Sundial: Fault-tolerant Clock Synchronization for Datacenters

Yuliang Li[*†], Gautam Kumar*[*], Hema Hariharan*[*], Hassan Wassel*[*], Peter Hochschild*[*], Dave Platt*[*],*
Simon Sabato[‡], Minlan Yu*[†], Nandita Dukkipati*[*], Prashant Chandra*[*], Amin Vahdat*[*]*
Google Inc.[*], Harvard University*[†], Lilac Cloud*[‡]

Abstract

Clock synchronization is critical for many datacenter applications such as distributed transactional databases, consistent snapshots, and network telemetry. As applications have increasing performance requirements and datacenter networks get into ultra-low latency, we need submicrosecond-level bound on time-uncertainty to reduce transaction delay and enable new network management applications (e.g., measuring one-way delay for congestion control). The state-of-the-art clock synchronization solutions focus on improving clock precision but may incur significant time-uncertainty bound due to the presence of failures. This significantly affects applications because in large-scale datacenters, temperature-related, link, device, and domain failures are common. We present Sundial, a fault-tolerant clock synchronization system for datacenters that achieves ∼100ns time-uncertainty bound under various types of failures. Sundial provides fast failure detection based on frequent synchronization messages in hardware. Sundial enables fast failure recovery using a novel graph-based algorithm to precompute a backup plan that is generic to failures. Through experiments in a >500-machine testbed and large-scale simulations, we show that Sundial can achieve ∼100ns time-uncertainty bound under different types of failures, which is more than two orders of magnitude lower than the state-of-the-art solutions. We also demonstrate the benefit of Sundial on applications such as Spanner and Swift congestion control.

1 Introduction

Clock synchronization is increasingly important for datacenter applications such as distributed transactional databases [12, 32], consistent snapshots [11, 16], network telemetry, congestion control, and distributed logging.

One key metric for clock synchronization is the *time-uncertainty bound* for each node, denoted as ε in this paper, which bounds the difference between local clock and other clocks. This concept is used by TrueTime in Spanner [12]. Spanner leverages TrueTime to guarantee the correctness properties around concurrency control and provide consistency in distributed databases. Another example is consistent snapshots, which are commonly used for debugging or handling failures in distributed systems. To ensure consistency among snapshots, each node needs to wait for its time-uncertainty bound (ε) before recording the states.

Traditional clock synchronization techniques provide ε at the millisecond level (e.g., <10ms in TrueTime [12]), which is no longer effective for modern datacenter applications with increasing performance requirements and ultra low latency datacenter networks (e.g., with latency around 5μs [25]). Today's applications can benefit significantly from submicrosecond-level ε. For example, FaRMv2 [32], an RDMA-based transactional system, observes the median transaction delay can drop by 25% if we improve ε from ∼20μs to 100ns. CockroachDB [3] can significantly reduce the retry rate when ε drops from 1ms to 100ns based on an experiment in [13].

Providing submicrosecond-level ε can also enable new network management applications. For example, with submicrosecond-level clock differences across devices, we can measure one-way delay, locate packet losses, and identify per-hop latency bursts [23, 24]. It also enables synchronized network snapshots [37] which are useful for identifying RTT-scale network imbalance and collect global forwarding state. Accurate one-way delay provides a better congestion signal to delay-based congestion control [17, 29] to differentiate between forward and reverse path congestion.

There are several systems that achieve submicrosecond-level clock precision. The state-of-the-art commercial solution on precise clock synchronization is Precision Time Protocol (PTP) [4]. PTP is widely available in switches and NICs [6,8,9]. Each switch or NIC has a hardware clock driven by an oscillator, generates timestamped synchronization messages *in software*, and sends them over a spanning tree to synchronize with other nodes. Normally, oscillator drifts stay within ±100μs per second and the devices synchronize every 15ms to 2 seconds [4, 8]. A recent proposal DTP [21] sends messages in the physical layer every few microseconds and can also achieve ∼100ns precision. Huygens [13] is a clock-synchronization system built in software that achieves <100ns

precision by using Support Vector Machines to accurately estimate one-way propagation delays.

While these works provide high clock precision under normal cases, the time-uncertainty bound ε grows to 10-100s of μs as datacenters are subject to a variety of failures. In large-scale datacenters, there are common temperature-related failures which affect oscillator drifts. There are also frequent link, device, and domain failures (i.e., a domain of links and devices that fail together) that affect the synchronization across nodes (see §3).

In this paper, we present Sundial, which provides ∼100ns time-uncertainty bound (ε) under failures including temperature-related, link, device and domain failures and reports ε to applications – two orders of magnitude better than current designs. Even in cases of simultaneous failures across domains, Sundial provides correct ε to applications. Sundial achieves this with a hardware-software codesign that enables fast failure detection and recovery:

Fast failure detection based on frequent synchronous messaging on commodity hardware: Sundial exchanges messages every ∼100μs in hardware without changing the physical layer. The frequent message exchange enables fast failure detection and recovery, and frequent reduction of ε. To ensure fast failure detection for remote nodes in the spanning tree, Sundial introduces *synchronous* messaging which ensures that each node sends a new message only when it receives a message from the upstream.

Fast failure recovery with precomputed backup plan that is generic to all types of failures: To enable fast failure recovery, Sundial controller precomputes a backup plan consisting of one backup parent for each node and a backup root, so that each device can recover locally. The backup plan is generic to different types of failures (i.e., link, device failures, root failures, and domain failures) and ensures that after failure recovery, the devices remain connected without loops. We introduce a new search algorithm for the backup plan that extends a variant of edge-disjoint spanning tree algorithm [35] but with additional constraints such as no-ancestor condition (the edge in the current tree cannot be a forward edge in the backup tree) and disjoint-failure-domain condition (no domain failure can take down both the parent and the backup parent for any device). Our algorithm only takes 148ms on average to run on an example Jupiter [33] topology with 88K nodes.

We evaluate Sundial with experiments in a >500 machine prototype implementation and via large-scale simulations. Sundial achieves ∼100ns time-uncertainty bound both under normal time and under different types of failures, which is more than two orders of magnitude lower than the state-of-the-art solutions such as PTP [4], Huygens [13], and DTP [21]. Sundial reduces the commit-wait latency of Spanner [12] running inside a datacenter by 3-4x, and improves the throughput of Swift congestion control [17] by 1.6x under reverse-path congestion.

2 Need for *Tight* Time-uncertainty Bound

A clock synchronization system for datacenters need not only a current value of time but also time-uncertainty bound that is used by applications for correctness as well as performance. We describe several datacenter applications and how tight time-uncertainty bound benefits them below.

Distributed Transactional Databases: Spanner [12], FaRMv2 [32] and CockroachDB [3] are some examples of distributed databases deployed at scale in production that directly use time-uncertainty bound to guarantee consistency – transactions wait out time-uncertainty bound before committing a transaction. Spanner is the first to use ε in production transactional systems. While it is globally distributed, its idea of using ε is adopted in many intra-datacenter systems such as FaRMv2 [32]. However, inside datacenters, with recent software and hardware improvements such as RDMA, NVMe, and in-memory storage, transaction latencies are going towards microsecond level. For example, FaRMv2 is built atop RDMA for datacenters and has ε of ∼20μs which already accounts for 25% of median transaction latency! Tight ε improves the performance of these systems both in terms of latency and throughput.

Consistent snapshots: Consistent snapshots [11, 16] is another important application for datacenters for debugging, failure handling, and recovery for cloud VMs. The consistency across servers can be guaranteed by waiting out ε to ensure the scheduled snapshot time is passed. With recent software and hardware improvements, ε becomes a performance bottleneck at a similar level as in distributed databases, limiting the frequency of taking snapshots.

Network telemetry: As network latency reduces to the order of a few microseconds, millisecond-level ε is too coarse-grained. Tight ε enables a wide range of fine-grained network telemetry. For example, per-link latency or packet losses can be measured by comparing the timestamps or counters at both ends of a link read at the same time [23, 24, 40]. Synchronized network snapshots at RTT scale can be enabled with tight time-uncertainty bound, and can be used for various telemetry needs such as measuring traffic imbalance across different links/paths in the dataceter [37].To achieve these, switch clocks also need to be synchronized.

One-way delay (OWD): Synchronized clocks enable the measurement of one-way delays. Small ε provides a tighter bound on the error in the measurement especially under failures. Measurement of OWD is useful for many applications including telemetry and congestion control. For example, OWD differentiates between forward and reverse-path congestion improving performance of delay-based congestion control algorithms such as Swift [17] (§6.3).

Distributed logging: A key challenge for debugging large-scale distributed systems is to analyze logs collected from different devices with clock differences. Tighter ε enables more useful analysis and opens up more distributed debug-

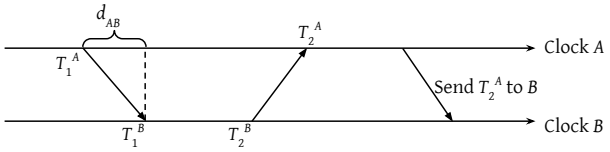


Figure 1: Message exchanges to synchronize B to A.

ging opportunities. Our ∼100ns ε is about the same as L3 cache miss time, so it can help order all log messages in a datacenter. We note that this class of applications has an additional requirement in that the synchronized clocks follow a master clock that reflects the physical time of day (§4.5).

3 Failures in Clock Synchronization System

In this section, we discuss the different failure scenarios affecting a clock synchronization system and their respective impacts. We start with a brief background on clock synchronization to aid the discussion.

3.1 Background on Clock Synchronization

The clock is driven by a crystal oscillator. Every device has a clock, whose value is incremented on every tick of a hardware oscillator. Different oscillators, even of the same type, have slightly different frequencies. The frequency of an oscillator may change over time, due to factors such as temperature changes, voltage changes, or aging resulting in clocks to drift away over time. As an example, oscillators in production networks can have a frequency variation of ±100 ppm (parts per million) [7], meaning that the oscillator can drift within the range of ±100μs per second compared to running at the nominal frequency. More stable oscillators (e.g., atomic clocks based on Cesium, Hydrogen or Rubidium particles or oven-controlled oscillators) are too expensive to deploy on every device in production.

Clocks exchange messages with each other for synchronization. To ensure that clocks remain close to each other, we need to periodically adjust the clocks to account for potential drift. Figure 1 shows an example where clock B synchronizes to A. A sends a synchronization message (abbreviated as sync-message in this paper) with a timestamp T_1^A based on A's clock, and B records the receiving time (timestamped by B) of the sync-message T_1^B. Now, if B knows the message delay d_{AB} from A to B, B can compute the *offset* between A and B as $T_1^A + d_{AB} - T_1^B$. To know d_{AB}, B sends another message to A to measure RTT, and use half of RTT to estimate: $d_{AB} = (T_2^A - T_1^A - (T_2^B - T_1^B))/2$. B uses *offset* to adjust its clock. A periodically sends out these sync-messages at an interval denoted by sync-interval. The accuracy of d_{AB} depends on multiple factors and we discuss them below.

A network of clocks synchronize using a synchronization structure. A common way to do this is to construct a spanning tree over which sync-messages are sent, e.g., PTP which is the most widely available system for datacenter clock synchronization uses a spanning tree with one device serving as the root (called master or grandmaster). The model for best case synchronization is that each device's parent is one of its direct neighbors in the physical network and sync-messages flow periodically from the root across the spanning tree.[1] This has two advantages. First, it allows switch clocks to also be synchronized enabling additional telemetry applications (§2). Second, it significantly improves the measurement of d_{AB} as shown in Figure 2. Noises in estimation of d_{AB} by halving the RTT can arise due to (1) asymmetric propagation delays of the forward path and the reverse path, and (2) queuing delays. For direct neighbors in the physical network, propagation delay asymmetry is near zero, and there is no queuing delay[2]. There are proposals that do not use a spanning tree as the synchronization structure but either they don't reflect the physical time [21] (§4.5) or they cannot provide submicrosecond-level precision [12, 27, 28] (§7).

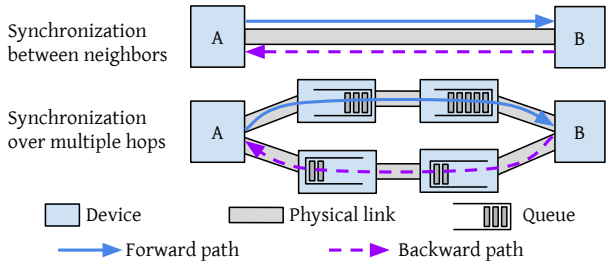


Figure 2: Benefit of synchronization between neighbors: symmetric forward and backward paths, and no noises from queuing delay.

Time-uncertainty bound. As clocks can drift apart over time, time-uncertainty bound (ε) can be calculated as:

$$\varepsilon = \varepsilon_{base} + (now - T_{last_sync}) \times max_drift_rate \quad (1)$$

ε of a clock exhibits a sawtooth function. T_{last_sync} is the last time when the clock is synchronized to the *root* (not just its direct parent), $now - T_{last_sync}$ increases with time and goes back to zero after synchronization to the root, and max_drift_rate is a constant representing the maximum possible drift rate between the local clock and the root's clock. The ε_{base} is a small constant (a few nanoseconds) that accounts for other noises (e.g., timestamping errors, bidirectional delay asymmetry of physical links, etc.).

We will show that in the face of failures in production environments, max_drift_rate should be conservatively derived (§3.2.1), and $now - T_{last_sync}$ can be large (§3.2.2).

3.2 Impact of Failures on ε

We classify failures affecting clock synchronization into three categories and study their impact on ε – failures that induce large frequency variations and need a conservative setting of max_drift_rate, connectivity failures that affect T_{last_sync}, and incorrect behaviors due to broken clocks and message corruption that need to be detected and addressed.

3.2.1 Failures that Induce Large Frequency Variations

An oscillator's frequency can incur a large variation in the event of sudden temperature or voltage fluctuation. Cooling failures are common and can affect thousands of machines.

[1] Note that PTP doesn't require this to be the case.

[2] While the devices may have local queues, the timestamp is marked at dequeue/egress time and is not subject to local queuing delay.

In an cooling incident that occurred in production recently, it resulted in errors related to clock synchronization in a large fraction of machines (and not just the ones affected by the failure). The temperature variation resulted in oscillator frequency variation to exceed *max_drift_rate* and the operator had to shut down many machines.[3] Thus, the *max_drift_rate* needs to be set very conservatively (e.g., 200ppm in TrueTime [12]) to tolerate frequency variations under a wide range of temperature (e.g., up to 85 °C) even though in normal cases, temperature variations occur slowly [13]. This entails that in order to keep ε small, we need to reduce $now - T_{last_sync}$ through frequent messaging – ε of 100ns with *max_drift_rate* of 200ppm needs sync-interval to be <500μs. Software cost of reducing sync-interval to such low values is high – PTP takes one core to process thousands of sync-messages and associated computations per second [1], and Huygens consumes 0.44% CPU for a sync-interval of 2s (which grows proportionally as the interval is reduced). We need hardware support for efficiency (§4.1).

3.2.2 Connectivity Failures

Failures that break the connectivity of the spanning tree also affect ε. For example, if a device or a link in the spanning tree fails, the whole subtree under this device or link loses connectivity to the root[4], until a new spanning tree is reconfigured by the SDN controller. ε grows proportionally to the time it takes for recovery – if it takes 100 ms, ε grows to more than 20μs. Even a distributed spanning tree protocol supported by PTP (best master clock algorithm) is slow to converge.

What is worse, is that the inflation of ε is not only for a device affected by the failure at a given time; instead, **almost all devices have to report high ε, all the time** and not only during the failure duration. This is because a device cannot distinguish whether it is affected by a failure or not. Consider a 3-node setup as depicted in Figure 3 with *A* as the root of the spanning tree and *B* and *C* as *A*'s child and grandchild respectively. When *A* fails, *B* detects the failure but *C* continues synchronizing to *B* without noticing the failure. This means at any time, there is no way for *C* to tell if it is in-sync or not, no matter if there is an actual failure or not and thus, it has to **always** report large ε (i.e., > 20μs) even during normal periods.[5] Another way to look at this is in the context of Equation 1, *C* cannot set T_{last_sync} to the time it receives the last sync-message from its parent T_{last_msg}; instead, for correctness, *C* has to **always** set $T_{last_sync} = T_{last_msg} - T_{recovery}$, where $T_{recovery}$ is the maximum time to recover from any failure that may break its connectivity to the root. All non-direct descendants of the root are affected by this.

[3] Normally, after a cooling system failure, operators let machines continue running for 10s of minutes before the recovery of cooling system or a gradual shutdown of machines, because this is usually safe and a sudden total shutdown should be avoided as much as possible.

[4] PTP is configured on a per-port basis (not per-device), so sync-message cannot bypass the failed link or the link associated with the failed device.

[5] Without changing the PTP standard, *B* cannot explicitly communicate to *C* about the failure.

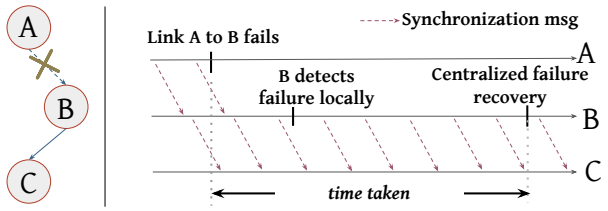


Figure 3: Challenge of determining T_{last_sync}. Node C cannot determine if it is synchronized to the root or not, so C has to always set T_{last_sync} conservatively early to account for possible down time.

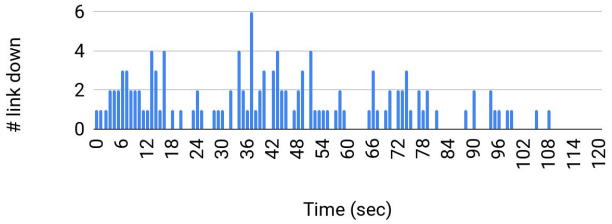


Figure 4: Number of link down events per second in a 1000-machine cluster during a near two-minute window of a failure incident.

There are many possible causes of connectivity failures: besides the common link or switch down, there are incidents that can take down massive (10s to 100s) devices or links, such as failures related to patch panels, link bundles, power domains, or human operations [38, 39]. Figure 4 shows the time series of link down events in a 1000-machine cluster during a failure incident. The suspected cause was a software bug related to a patch panel but its impact on device/link failures lasted across nearly two minutes – a total of 133 links go down. Thus, in order to provide small ε, the system must recover from connectivity failures quickly.

3.2.3 Broken Clocks and Message Corruption

Clocks may break and stop functioning well resulting in actual drift rate to exceed *max_drift_rate*. While this is rare relative to more severe hardware problems – statistics from production show that broken CPUs are 6 times more likely than broken clocks [12] – they need to be taken care of to provide correct ε to applications. Similarly, sync-message corruption may garble the associated timestamp and affect correctness of reported ε. A fault-tolerant clock synchronization system must detect and address such anomalies.

4 Sundial Design and Implementation

Motivated by the discussion above, we identify two key requirements to build a fault-tolerant clock synchronization system for datacenters that achieves performant time-uncertainty bounds. First is a small sync-interval (§3.2.1) – this is well served with a hardware implementation to avoid high CPU overhead of receiving and transmitting synchronization messages in software. Second is fast failure recovery so that ε continues to be small even when failures happen (§3.2.2). The challenge here is that recovering solely via a centralized controller is slow for our target ε requirements. Instead, as we show later, we can recover from most failures locally by adding redundancy to the synchronization graph, where in

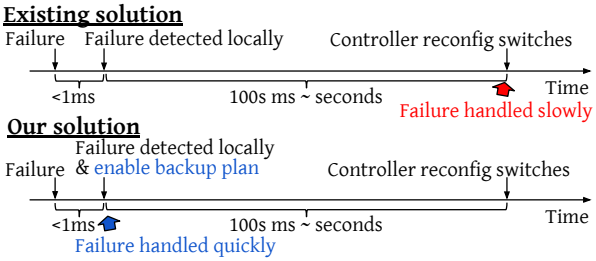


Figure 5: Fast failure recovery using precomputed backup plan.

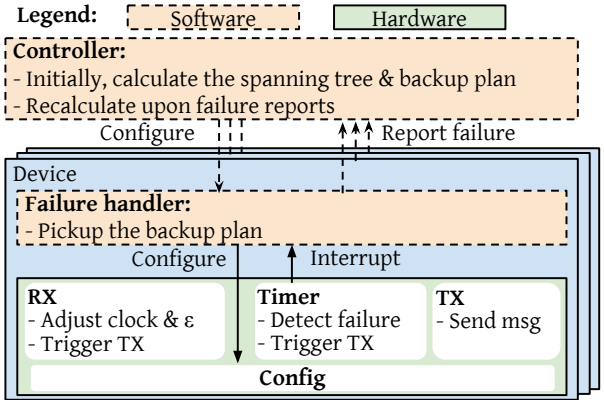


Figure 6: Sundial Framework. Solid arrows are the fast local recovery. Dashed arrows are slower but non-critical paths of recovery.

addition to the primary spanning tree, each device maintains a backup parent, such that it can transition to the backup parent locally upon detecting a failure. As shown in Figure 5, this takes the round trip time to the controller and the computation time out of the critical path of failure handling.

Thus, Sundial uses a hardware-software codesign. Figure 6 depicts Sundial's framework, which has three main components. Sundial implement the most essential functions of exchanging synchronization messages and detecting failures in hardware such that it can synchronize frequently and quickly detect failures. Sundial relies on software components to take action once a failure is detected, by invoking a failure handler in software which reconfigures the hardware to transition to the backup parent pre-programmed by a centralized controller (also in software). We use the topology in Figure 7(a) as a toy example to aid with the discussion in this section with Figure 7(b) as an example spanning tree.

4.1 Sundial Hardware Design

Sundial's hardware has three main components. It implements *frequent* transmission of sync-messages in a *synchronous* fashion, i.e., sync-messages are sent downstream only upon their receipt. The hardware is also responsible for detecting failures and triggering software handlers for quick recovery. Finally, the hardware maintains the current value of ε. We detail out these components below.

4.1.1 Frequent Synchronous Messaging

Sundial's hardware supports frequent message sending to prevent clocks from drifting apart significantly. On the root, this is done via a hardware timer maintaining a counter that

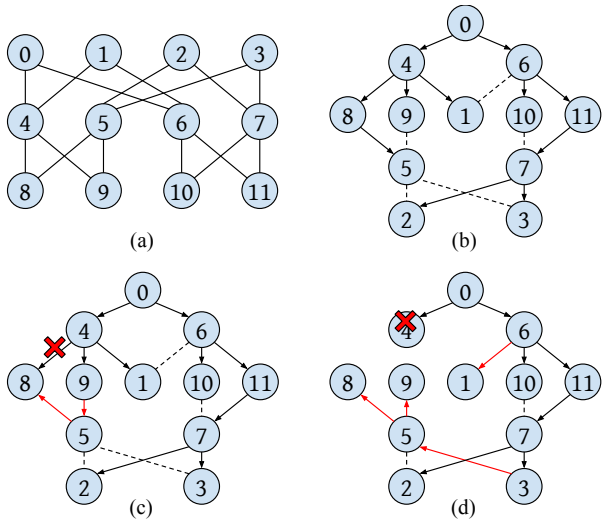


Figure 7: Failure cases in a k=4 FatTree. (a) is the raw FatTree. To show the spanning tree clearer, we draw an equivalent topology in (b) and a spanning tree in it. An arrow is from a parent to its child, and a dashed line indicates an edge not used in the spanning tree. (c) shows one way of adjusting the spanning tree when the link between 4 and 8 fails; not only the directly impacted nodes (node 8), but also other nodes (node 5) have to change parent. (d) shows one way of adjustment when node 4 fails; the way node 5 changes its parent (to node 3) is different from the case in (c) (change to node 9).

increments on every oscillator cycle, and triggers message transmission when the time since last transmission exceeds the configured sync-interval. We configure sync-interval on the root device to be around 100μs. The sync-messages are sent at the highest priority, but the network overhead remains extremely small – a 100-byte packet every 100μs only consumes less than 0.01% bandwidth and adds at most 10ns queuing delay for other traffic.

For non-root devices, a challenge is that an upstream failure can affect all devices in that subtree. Consider the case in Figure 7(c), if link 4-to-8 goes down, 8 needs to switch to 5 as its parent, which needs 5 to change its parent as well. A potential solution is explicit notification of failures to other devices, but this has two issues – not only can this be unreliable (since the notification messages may get dropped), it also adds complexity to the hardware. Instead, we solve this via *synchronous messaging* where message transmission is triggered only upon receipt of a message from upstream. In this way, an upstream failure implies that messages stop propagating downstream, and devices can take corrective actions.

4.1.2 Fast Failure Detection

Sundial's hardware uses a timeout to detect if it stops receiving messages indicating an upstream failure. The timeout is set to span multiple sync-intervals, such that occasional message drop or corruption doesn't trigger it. It's implemented using a counter that is incremented on every oscillator cycle, and cleared on receiving a sync-message – once it's exceeded, the hardware issues an interrupt to the software.

To detect broken clocks and message corruption, each de-

vice verifies the incoming timestamp (adjusted for link delay). If the adjusted value lies outside the local ε, the message is marked invalid and doesn't trigger an update and message transmissions. A broken clock can cause continuous invalid messages and thus, we don't reset the timeout counter on their receipt. Once a broken clock is detected, the failure handler in device software is triggered to handle it (§4.2.2).

4.1.3 Time-uncertainty Bound Calculation

The hardware maintains ε according to Equation 1. In our implementation, we configure max_drift_rate = 200ppm and ε_{base} = 5ns×$depth$ where $depth$ is the distance of the device from the root in the tree.

T_{last_sync} is updated when receiving a sync-message. In PTP, T_{last_sync} should be set to earlier than T_{last_msg}. Thanks to synchronous messaging, Sundial sets it to T_{last_msg} since a device stops receiving messages on an upstream failure. This lowers $now - T_{last_sync}$ which in turn lowers ε.

4.2 Sundial Software Design

There are two main components to round out the fault-tolerant design of Sundial – a centralized SDN controller that precalculates backup plans and programs them on the devices and a failure handler in device software that quickly moves to the backup when a failure is detected by the hardware.

4.2.1 Centralized Controller

The centralized controller in Sundial is responsible for computing the primary spanning tree along with the backup plan based on the current topology and configures the devices accordingly. Comparing Figure 7(c) and 7(d), we see that not all neighbors of a node (e.g., node 5 in the figure) can be the backup parent under different failures. Sundial uses a search algorithm (detailed below) to compute a fault-tolerant backup plan that is *generic* to link, non-root node, root node, and domain failures (which can take down multiple links or devices). We break down this requirement into 5 properties.

Properties of a fault-tolerant backup plan. We briefly introduce the terminology used. The **primary spanning tree** is one that is currently being used to propagate sync-messages. The **backup plan** consists of a backup-parent for each node/device and a backup root. Terms like parent, edges, paths, and ancestors apply separately to the primary and the backup graph (graph formed by the edges in the backup plan).

(1) No-loop condition: *For any primary subtree, the backup edges of nodes in the subtree do not form a loop.* This is a necessary and sufficient condition to be generic to any single link failure. The necessity is obvious: if there is a loop, the nodes in the loop do not synchronize to the root after a failure. We prove the sufficiency by induction as follows. Suppose a k-node subtree is affected by a link failure, and the k backup edges do not form a loop (Figure 8); the nodes other than the k nodes are unaffected and still form a tree (called the *main tree*). At least one of the k nodes' (say, C) parent is in the main tree; otherwise, all k nodes' parents are in the k nodes, which must form a loop, contradicting the no-loop condition.

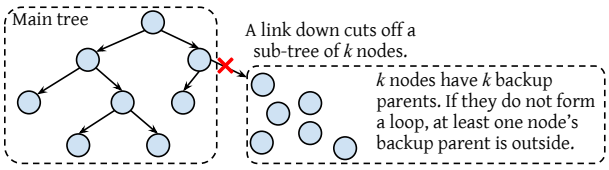


Figure 8: No-loop condition. It is sufficient to guarantee connectivity after any link failure.

We can now expand the main tree to include C since C is connected to the main tree via its backup edge. We can then iteratively add the remaining $k-1$ nodes to the main tree.

(2) No-ancestor condition: *The backup parent of a node is not its ancestor.* This and property (1) together ensure that the backup plan is generic to any non-root node failure. Otherwise, if that ancestor fails, that node has no backup parent.

(3) Reachability condition: *The backup root must be able to reach all nodes through backup paths.* This is necessary and sufficient to be generic to the root failure. When the root fails, all nodes change to their backup parents, and the backup root will become the new root. To synchronize all nodes, they must be reachable from the backup root.

(4) Disjoint-failure-domain condition: Domain failures present a unique challenge, because they may take down multiple devices or links. If a domain failure breaks the connectivity of a device s to the root, s will turn to its backup parent; but if the domain failure also takes down its backup parent, then s cannot recover its connectivity.

The following property solves this problem: *for any node s, there shouldn't be a single domain failure that both breaks s's connectivity to the root and takes down the backup parent or backup edge, unless that failure also takes down the node s.*

Formally, if the set of failure domains that can break s's connectivity to the root[6] is D_p, the set of failure domains that can take down s's backup parent or backup edge is D_b, and the set of failure domains that s belongs to is D_s, we should have $D_p \cap D_b \subseteq D_s$.

The necessity is obvious. We present the intuition behind the proof of the sufficiency. If a domain failure happens, s has two possibilities: either s's connectivity is unaffected, or s connects to its backup parent b. If it is the latter, then the questions is whether b is connected to the root, which also has two possibilities. Doing this recursively, s keeps connecting to more nodes along a backup path. The backup path will not go indefinitely due to the no-loop condition, so it finally reaches either an unaffected node or the root.

(5) Root failure detection: Upon root failure, the backup root needs to collect sufficient information to elect itself. Figure 9 describes the approach – the backup root is chosen amongst root's children so it has one source of information by itself.

To get information from additional sources, we set up the backup graph to have a backup path from the subtree of another child of the primary root (i.e., the backup path from node

[6] Any device or link failure along the primary path from the root to s can break s's connectivity.

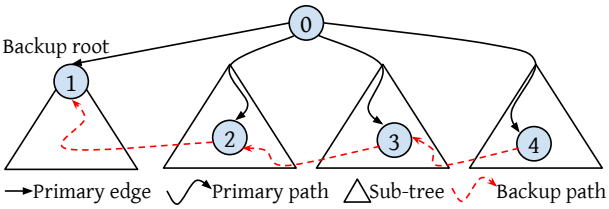


Figure 9: Root failure detection. Under any non-root failure, the backup root continues receiving messages, which can be used to distinguish other failures.

2 to 1 in Figure 9). In this way, if the link between the primary root and the backup root fails (link from 0 to 1), the backup root knows the primary root is still alive because it continues receiving sync-messages that come through the backup path. We can continue this backup path to cross more subtrees of children of the primary root to get additional sources of information (e.g., crossing node 3 and 4 in Figure 9).

In this way, as long as the root is alive, the backup root continues receiving sync-messages. Only when the root fails, the backup root stops receiving messages. So the backup root can detect the primary root failure using a second timeout of not receiving messages after it first turns to its backup parent, and it elects itself as the new root after the second timeout.

Putting all 5 properties together. Only non-root nodes have backup parents, so there are N-1 nodes and N-1 edges in the backup graph (N is the total number of nodes), so there must be exactly one loop[7] in the backup graph, and each node in the loop has a backup subtree (can be a single node) under it (Figure 10). With property (3), the backup root must be in the loop, so that the backup root can reach all nodes. The loop should cross multiple primary subtrees of root's children, so it meets both property (1) and property (5) (it delivers multiple sources of primary root's information to the backup root). Lastly, the backup graph should meet properties (2) and (4).

Figure 11(a) shows an example of primary tree and backup graph for the topology in Figure 7. Note that the computed primary tree is different to support a backup graph. The backup graph has a loop (between node 4 and 8) with the backup root 4 on it; the loop crosses the two primary subtrees of root's children (node 8 is under node 6's primary subtree). To show how property (4) handles domain failures, we add a failure domain that includes both node 11 and 3 (primary and backup parents of node 7 in Figure 11(a)). Now in the new backup graph (Figure 11(b)), to meet property (4), node 7's backup parent becomes node 2, so that even if both node 3 and 11 go down, node 7 (and other nodes) is still connected.

We want to highlight how the system recovers when the root fails. All backup edges get enabled forming a loop, but no sync-messages flow at this time. At the second timeout, the backup root elects itself and ignores incoming messages, effectively disabling the edge towards it (Figure 10). In this way, sync-messages do not loop.

[7] A graph with equal numbers of nodes and edges has at least one loop. In addition, if there is more than one loop, then the graph is not fully connected.

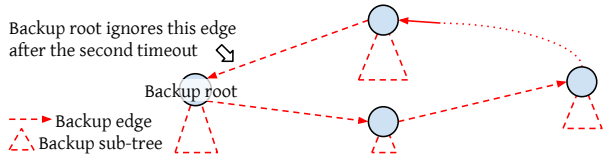


Figure 10: Backup Graph. There is exactly one loop with the backup root in it. Each node in the loop is the root of a subtree.

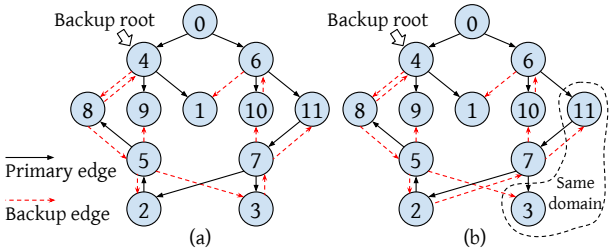


Figure 11: (a) A primary tree and a backup graph that meet all properties in Figure 7. But if node 3 and 11 are in the same domain, node 7 cannot have them as its primary and backup parents, so its backup parent becomes node 2 in (b).

Algorithm for computing backup plan. Sundial uses a search algorithm to calculate the backup plan which includes a primary tree and the backup graph. Note that not every primary tree has a valid backup graph. Thus, the goal is to search for a primary tree and its backup graph together. The search heuristic is based on the *score* of a primary tree – the maximum number of edges in the backup graphs it supports. The corresponding backup graphs are called the largest backup graphs (of the primary tree).

Algorithm 1 describes the algorithm. *pending* is the set of primary trees that are pending to be checked, initialized with a simple BFS (Line 1). After initialization (Line 2), we start the SEARCH function (Line 3) that will return a pair of primary tree and backup graph. In SEARCH, each time, we pick the primary tree p with the highest score (Line 6) – the most promising one – and find the largest backup graphs for it (Line 7). If some backup graph is complete, i.e., every device (including the backup root) has a backup parent, the search successfully returns (Line 8 - 9). Otherwise, we update the best score so far (Line 10), and mutate p (Line 11) to get a new set of primary trees in *pending* and iterate.

In MUTATE, for each backup graph b (Line 14), we try to expand b to include edge $<x, y>$ (Line 15). Since $<x, y>$ is not usable in backup graphs of p[8] (i.e., USABLEINBACKUP($<x, y>$, p) is false), we IMPROVE p to make $<x, y>$ usable (Line 16). We then add each improved version p' to *pending* if not already tested (Line 19). After all the mutations, p is removed from *pending* (Line 22). We will discuss the optimizations in Line 20 - 21 later.

FINDLARGESTBACKUP and IMPROVE are the key functions. FINDLARGESTBACKUP conforms to the 5 properties. Properties (2) and (4) decide what edges can be used in backup graphs given a primary tree p, as expressed in function

[8] $<x, y>$ is not usable for sure; otherwise b is not the largest because it can readily include $<x, y>$.

Algorithm 1 Searching for a primary tree and a backup graph.

```
1:  pending = {BFS(prim_root)};
2:  tested = ∅; best_score = 0;
3:  return SEARCH();
4:  function SEARCH
5:      while pending is not empty do
6:          p = pending.get_best(); tested∪={p};
7:          backup_set = FINDLARGESTBACKUP(p);
8:          if ∃b ∈ backup_set|b is complete then
9:              return p, b;
10:         best_score=max{best_score,calc_score(p)};
11:         MUTATE(p, backup_set);
12:     return NotFound;
13: procedure MUTATE(p, backup_set)
14:     for b in backup_set do
15:         for each <x, y> | x ∈ b, y ∉ b do
16:             new_prim_set=IMPROVE(p, <x, y>, b);
17:             for p′ in new_prim_set do
18:                 if p′ ∉ tested then
19:                     pending∪={p′};
20:                     if calc_score(p′)>best_score then
21:                         return ;
22:     pending-=p;
```

Algorithm 2 Check if <x, y> is usable in backup graphs of p.

```
1: function USABLEINBACKUP(<x, y>, p)
2:     return (x is not y's ancestor in p) && (y's ancestor in p and
       x meet disjoint-failure-domain condition);
```

USABLEINBACKUP (Algorithm 2). Properties (1), (3), and (5) decide how the backup graph should look like. We can simply use BFS starting from the backup root (property (3)) to find the tree (property (1)) that is largest, and then enumerate the backup parent for the backup root and see if it forms a loop that meets property (5). IMPROVE's goal is to change p to p' so that <x, y> becomes usable (i.e., USABLEINBACKUP(<x, y>, p') is true). It finds the set of p' that meets this goal.

As long as FINDLARGESTBACKUP and IMPROVE are exhaustive, the search is exhaustive – it will find a solution if one exists. The search process is similar to an algorithm that finds two edge-disjoint spanning trees [35], because our backup graph is composed of a more restricted spanning tree that is edge-disjoint with the primary tree, and an extra edge towards the backup root. The problem seems to be NP-hard although we don't have a proof yet.

In practice, our implementation of SEARCH is extremely fast – it only takes 148ms on average in a simulated Jupiter topology with 88,064 nodes [33] leveraging the following three strategies. (i) In Line 20 - 21 of Algorithm 1, we prune enumerations as per Line 14 - 15 as long as we find a p' that is heuristically better than any primary trees so far (including p). This significantly speeds up the search, as we can immediately make progress – after return (Line 21), the search immediately starts a new iteration at Line 6 based on p', which is heuristically better than continuing mutating p. Note this strategy does not miss any primary trees, as the original p remains in *pending*. (ii) FINDLARGESTBACKUP only returns one of the largest backup graphs, rather than all of them. This is sufficient as all largest backup graphs for a primary tree connect the same set of nodes and we use this strategy in Algorithm 3. (iii) IMPROVE just returns the set of p' that *keeps the largest number of b's edges usable*. This tries to keep as many useful fruits of past iterations as possible, so it speeds up the search. Algorithm 4 is based on this strategy.

Algorithm 3 Finding the largest backup graph of p.

```
1: function FINDLARGESTBACKUP(p)
2:     b=BFS_ForBackup(backup_root, p);          ▷ BFS uses
       USABLEINBACKUP to avoid unusable edges.
3:     Find <y, backup_root> where y ∈ b && USABLEIN-
       BACKUP(<y, backup_root>, p) && the loop crosses multiple
       subtrees of prim_root in p; Add <y, backup_root> to b;
4:     return {b};
```

Algorithm 4 Changing p to make <x, y> usable and keep as many b's edges usable as possible.

```
1:  function IMPROVE(p, <x, y>, b)
2:      if x is y's ancestor in p then
3:          for each edge <u, v> on the path x ⇝ y in p do
4:              new_prim_set∪ = RECONNECT(v, x, p, b);
5:      if <x, y> fails disjoint-failure-domain condition then
6:          new_prim_set∪ = RECONNECT(y, x, p, b);
7:      return new_prim_set;
8:  function RECONNECT(v, x, p, b)
9:      BFS from v along reverse edges, and stops at nodes outside
       x-subtree in p, while keeping as many b's edges usable as pos-
       sible. It gives a set of paths S={w ⇝ v|w is outside x-subtree in
       p}
10:     for path in S do
11:         For each <i, j> ∈ path set j's parent to i in p to get p′;
12:         new_prim_set∪ = {p′};
13:     return new_prim_set;
```

These three strategies significantly reduce the computation time per iteration (Line 6 - 11). While the latter two strategies make the search non-exhaustive, all practical datacenter topologies have high redundancy such that in our experiments, we quickly found a backup-plan even after injecting 50 successive failures. Also owing to the high redundancy in practical topologies, the number of iterations is small since the initial p already has a very high score, only a few hundreds below the total number of nodes. Finally, another consequence of high redundancy is that in practice, the search iterates with almost monotonically increasing scores [9], sometimes with jumps of tens or hundreds, reaching the final backup plan in tens of iterations on average.

Mutation for meeting property (5) follows a similar process as MUTATE. We omit the details due to limited space.

Calculating $\varepsilon_{base,backup}$. When a node turns to its backup parent, its *depth* may change, so we also precompute $\varepsilon_{base,backup}$

[9] IMPROVE can easily find paths while keeping all b's edges usable, because of the high redundancy.

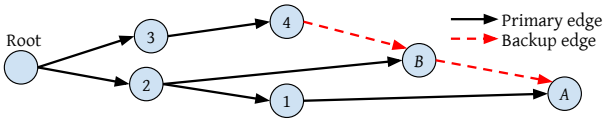


Figure 12: Node A's *depth* is dependent on the failure. If node 1 fails, A's *depth* is 3 (A, B, 2, root). But if node 2 fails, A's *depth* is 4 (A, B, 4, 3, root).

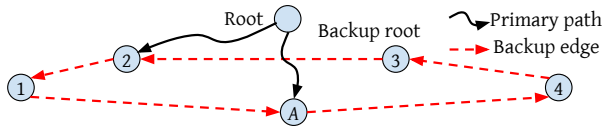


Figure 13: A has 6 possible paths to the root, of 3 types. **(1) Backup path**: if the root is down, the backup path (A, 1, 2, 3) is in effect. **(2) Primary path**: when A is unaffected by failures. **(3) Mixed path**: when failures affect A and some other nodes on the loop, A connects to the root first along the loop for one or more hops, and then along the primary path (e.g., A, 1, 2, ..., root). There are 4 possible mixed paths, starting the primary paths from respective node 1, 2, 3, and 4.

to which a device set ε_{base} upon timeout. The exact *depth* is failure-dependent as shown in Figure 12.

So we calculate the maximum possible *depth* for each node after *any* failures. A naive approach is to enumerate all possible combinations of failures, which can be slow. Instead, Sundial uses a simple dynamic-programming (DP) based scheme. If a node s turns to its backup parent b, we calculate s's maximum possible depth $s.depth_{backup}$:

$$s.depth_{backup} = 1 + \max(b.depth_{primary}, b.depth_{backup})$$

where the max function considers two possible cases: b is unaffected by failures ($b.depth_{primary}$ denotes b's depth in the primary tree, a deterministic value), or affected by failures. $depth_{backup}$ can be calculated top-down.

DP works for all nodes except the nodes on the loop in the backup graph, whose DP calculations inter-depend. But we can easily calculate their maximum possible depths. On an L-node loop, for each node we enumerate all $L+1$ possible ways it connects to the root (Figure 13). So the overall time complexity is $O((N-L)+L(L+1))$ for a total of N nodes.

4.2.2 Failure Handler in the Device Software

A daemon running in firmware serves as the failure handler and responds to interrupts generated by the hardware once it detects a failure – the hardware is reconfigured to move to the backup parent based on the backup plan and set ε_{base} to $\varepsilon_{base,backup}$. For the backup root, if an interrupt is triggered, the failure handler also continues to monitor incoming sync-messages for the second timeout. At the second timeout, the device sets itself as the primary root.

Handling broken clocks. If a clock is broken [12], it can drift away faster than *max_drift_rate*. In Sundial, we detect such clocks in two steps: (1) detect the existence of a broken clock when receiving an invalid message, and (2) confirm which one is broken. Figure 14 illustrates the process. As such, a broken clock is isolated without affecting other clocks.

The failure handler is triggered by a hardware interrupt upon receiving an invalid message to handle broken clocks.

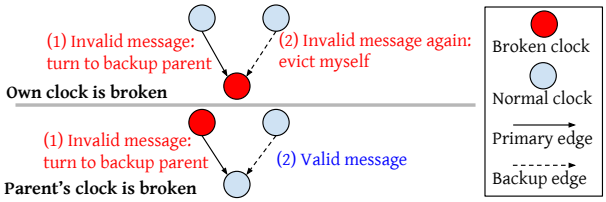


Figure 14: Handling a broken clock in two steps. If a node's own clock is broken, the messages from both its primary and backup parents are marked invalid by itself (the timestamp is outside local ε), so it evicts itself. If a node's parent's clock is broken, after receiving an invalid message it turns to its backup parent, and continues synchronization thereafter.

For the node with a broken clock, it evicts itself (no longer participates in synchronization). For the node whose parent has a broken clock, it turns to its backup parent.

4.3 Implementation

Controller. We implement a module in the network controller. The module registers a function to be called by the controller framework for failure notifications. When notified, this module reads the current device/link/port states, and computes a new backup plan. For each device, it compares the existing configuration and the new configuration, and only reconfigures the devices whose configuration changes, through RPC. It also configures the TX side of both primary and backup edges to send sync-messages.

RPC Interface between the Controller and Device Firmware. The controller and the device firmware communicates through RPCs. These RPCs have the following parameters: backup parent, first timeout, and second timeout which are used to configure the device hardware.

Firmware. The RPC handler configures the backup parent, the first timeout, and the second timeout accordingly. The backup parent and the second timeout are maintained in the firmware, and the first timeout is maintained in the hardware registers to enable failure detection in hardware. Only the backup-root has a non-zero value for the second timeout.

The firmware also registers a handler function for the interrupt triggered by the first-timeout. This function first reconfigures the hardware to accept sync-messages from the backup parent; then, if the second-timeout is non-zero, it waits for the timeout to see if it receives any new sync-messages; if not, it configures the hardware to become the root.

We cannot reveal hardware details due to confidentiality.

4.4 Practical Considerations

Concurrent connectivity failures may happen in practice, and may not be recovered by the backup plan, which needs to involve the controller. Sundial maintains the correctness of ε in this case. The only impact is that ε grows larger before being recovered by the controller: if it takes 100ms to recover, ε grows up to 20μs during this time (still $\sim$100ns during normal time). The impact is negligible, because compared to single failures, concurrent failures are already rare, and only a very small subset of them cannot be recovered by the backup

plan, as discussed below.

The most commonly seen concurrent failures are caused by a domain failure, which is not an issue because of the disjoint-failure-domain condition of the backup plan (§4.2.1).

If cross-domain failures happen, whether they impact Sundial depends on their locations and time proximity. For the nodes whose connectivity is affected, the backup plan is ineffective only if these failures also take down their backup parents/edges (special locations) within a short period of time before the controller recomputes a new backup plan (time proximity). The chance is very small, because cross-domain failures are random in locations and time proximity.

Small window of error before evicting a broken clock. The broken clock detection only happens when messages arrive. There is a small time window between when the failure actually happens and when the next message arrives, during which errors could arise. This can be solved via hardware redundancy – each node physically keeps two clocks, and each clock query reads the two clocks and checks if they match (their time-uncertainty ranges overlap). Once a clock is broken, the next read immediately detects it. Additionally, Sundial prevents this failure to affect other clocks, because its children ignore the invalid messages.

False positives. If a device timeouts without a failure, it will turn to the backup parent. Such false positives are harmless, except extra controller processing. We do not observe false positives in our experiments.

4.5 Sundial's Position in the Design Space

4.5.1 Design Space of Clock Synchronization

At the submicrosecond level, Sundial is the first to support time-uncertainty bound. We identify three key aspects of the design that a clock synchronization system must answer.

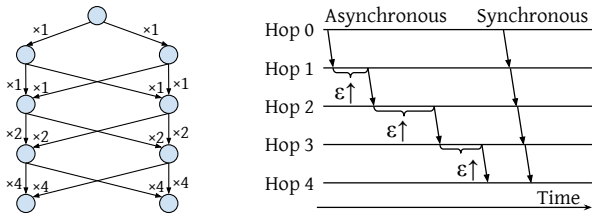
1. Type of message: There are multiple options, synchronization messages can either be sent directly with specialized physical layer (PHY) with zero-overhead messages, or at higher layers (L2, L3, L4) with increasing bandwidth overhead and increasing ease of deployment.

2. Noise due to message delay between a pair of clocks. The message delay in the forward and reverse directions may be unequal due to queuing or asymmetric paths. There are three options to deal with such noise: (1) Only synchronize between neighboring devices, such that there is no noise (§3.1). (2) Use multiple messages to filter out noise; (3) Tolerate the noise. Option (1) is the best if all devices (switches and hosts) can participate. Otherwise, option (2) and (3) face a tradeoff between noise and overhead.

3. Network-wide synchronization structure: three options.

(1) Master clock distributed through a tree. A master clock distributes its time to other clocks through a tree. The master clock can synchronize to the physical time (e.g., via GPS), so that all clocks reflect the physical time.

(2) Master clock distributed through a mesh. Similar to (1), but instead of a tree, each clock receives sync-messages from multiple other clocks, forming a mesh.



(a) Synchronous messaging: exponential inflation of sync-messages. **(b)** Asynchronous messaging has much larger ε.

Figure 15: Mesh structure: higher ε due to asynchronous messaging.

(3) No master clock (no physical time). Clocks synchronize independently with each other without regards to a master clock. For example, in DTP [21] each clock follows the fastest of its neighbors. In this option, all clocks converge to a function (e.g., max() in DTP) of all clocks, which has nothing to do with the physical time. This option is worse than (1) and (2) because access to physical time is important for many datacenter applications.

Tradeoff between (1) and (2). While (2) is clearly more fault-tolerant, it cannot get ε as low as (1). The reason is that mesh-based solutions cannot use synchronous messaging. As shown in Figure 15a, if a clock receives sync-messages from k other clocks, synchronous messaging inflates the number of messages by k per hop, causing exponential inflation. So mesh-based solutions have to use asynchronous messaging, which has much larger ε – as shown in Figure 15b, ε increases per hop from the master to the participant clocks. On the other hand, tree-based solutions can use synchronous messaging, achieving much lower ε. §6.2 evaluates this effect.

4.5.2 Sundial's Design Choices

Sundial's key contribution is in the third design choice, which exhibits fundamental tradeoff between small ε and fault-tolerance. Sundial aims to achieve the best of both worlds, by combining tree and mesh topologies: Sundial sends messages through a mesh, such that it still has available edges upon failures; but the effective synchronization only happens over a primary tree, enabling it to use synchronous messaging.

The first two design choices have clear best options, and they are mainly determined by hardware availability. In our implementation, Sundial synchronizes neighboring devices at the L2 level as the specialized PHY layer is not available. That said, Sundial can benefit from such a layer if it's available. Comparison with other schemes is in §7.

5 Application Access to Synchronized Clocks

In Sundial, the primary mechanism to access synchronized clocks is via hardware Rx/Tx timestamps. Additionally, for applications that want to access host clock directly, Sundial provides local host to NIC clock synchronization.

Access via hardware timestamps. NIC and switch hardware timestamps marked on the packets [29] are the primary access mechanism, for which it provides ∼100ns time-uncertainty bound. Applications such as distributed databases

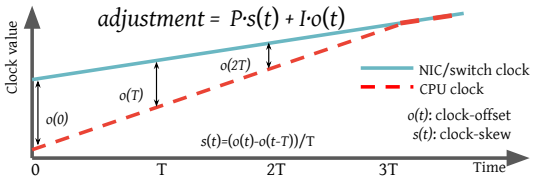


Figure 16: PI controller based on clock-skew; offset and skew are measured periodically and an adjustment is computed using suitable P and I constants.

that have strict ε requirements rely directly on NIC-Rx-timestamps marked on the last packet in a message to order them to provide consistency properties. Networking stacks such as Snap [26] provide op-stream interface to applications (preventing out-of-order delivery) and export the NIC timestamps. Telemetry and congestion control applications also rely directly on NIC timestamps to measure one-way delays.

Host clock synchronization. We also synchronize the local host clock to the NIC clock for applications that want to directly read the host clock (and don't require strict guarantees on ε). We use a Proportional-Integral controller based on clock-skew between the host and NIC clocks as depicted in Figure 16. We measure the offset, $o(t)$ and skew, $s(t)$, every T time-units (we use T=10ms) and apply the rate adjustment to the host clock to tick faster or slower. The constants P and I need to be tuned in production. One challenge is that the two clock-measurements are subject to local delays such as PCIe jitter and we use linear regression to filter the noise out.

6 Evaluation

Through experiments in a >500-machine testbed-prototype (§6.1) and through large-scale simulations (§6.2), we show that Sundial's time-uncertainty bound is ∼100ns under different types of failures, and discuss application improvements enabled by Sundial in §6.3.

6.1 Time-uncertainty Bound (ε) in Testbed

6.1.1 Methodology

Testbed. The testbed consists of 23 pods, 276 switches and 552 servers. A pod including 12 switches and 24 servers acts as a failure domain. The oscillators used in the hardware have a frequency specification of ±100ppm. The depth of the base spanning tree in the topology is 5.

Schemes for comparison. We compare Sundial with recent submicrosecond-level clock synchronization schemes: PTP [4], Huygens [13], and DTP [21]. While they do not consider time-uncertainty bound (ε) and how it is reported to applications, we augment the designs to provide ε, according to Equation 1 in §3.1 and describe them below.

***Sundial*:** We set the sync-interval to 90μs.[10] The timeout is 185μs (>2×sync-interval). The second timeout for the backup root to elect itself is set to 180μs (185+180>4×sync-interval). The backup plan has a maximum $depth_{backup}$ of 6.

***PTP+ε*:** PTP is the most common submicrosecond-level synchronization protocol with a default sync-interval of two seconds. To add ε, we set ε_{base} to 5ns×$depth$, and max_drift_rate to 200ppm. T_{last_sync} is updated as follows – for root's children, we set $T_{last_sync} = T_{last_msg}$; but for other descendants, we set $T_{last_sync} = T_{last_msg} - T_{recovery}$ to account for possible out-of-sync duration caused by remote connectivity failures that are oblivious to them (§3.2.2). We set $T_{recovery}$ to 2s, since it takes 2s to recover from failure.[11]

***PTP+DTP+ε*:** What if we could set lower sync-interval in PTP+ε? We evaluate another scheme that leverages DTP – DTP allows very small sync-interval (a few microseconds) with low bandwidth overhead by modifying the physical layer protocol. Since DTP requires hardware support, we emulate it in our testbed by setting 5μs sync-interval (much smaller than 90μs).[12] All devices that are not direct children of the root set $T_{recovery}$=100ms, where 100ms is the typical connectivity failure recovery time measured from datacenters.[13]

***Huygens+ε*:** Huygens gathers network-wide sync-messages during each 2-second sync-interval, and uses machine learning to decide the best adjustment for each device at the beginning of the next sync-interval. While we do not have its implementation, we report the best possible ε it can achieve. Specifically, we assume it is not affected by connectivity failures because of its use of network-wide information, so T_{last_sync} is set to the beginning time of each sync-interval (without minus $T_{recovery}$). We also assume it can filter out delay noises entirely and optimistically set ε_{base} to 0.

Failure injection. We evaluate the impact of failures on ε in Sundial and above schemes by injecting link failures, non-root device failures, root failures, and domain failures (where multiple devices can go down).

Metrics and measurement approach. We measure ε on every device by running a daemon in the firmware to read ε every 10μs. After a failure, the controller sends an RPC to configure the devices for recovery. The frequent monitoring interferes with processing RPCs that are sent by the controller in the event of failures. As a workaround, we set a stop time which allows the controller RPC to execute after the monitoring stops. In this way, the monitoring tells us which devices are affected by failures and their ε. But it also inflates the controller delay, which is unfair to other schemes as they heavily rely on the controller for failure recovery. With knowledge of the expected controller delay, we can easily restore the expected ε based on the measured ε (Figure 17), because ε's behavior is deterministic during failures recovery: ε keeps increasing, and goes back to normal when the failure is recov-

[10] 90μs is just enough for ∼100ns ε, although lower ε is achievable.

[11] In favor of low ε, $T_{recovery}$ = 2 seconds is already a very optimistic setting for PTP+ε, because recovery may take longer if the next sync-message is also dropped by another failure that just happens at that time. Setting $T_{recovery}$ larger results in even higher ε. But we show that even with this optimistic setting, PTP+ε still has much higher ε than Sundial.

[12] This is sufficient to show the improvement of ε, even though we don't have the physical layer protocol to keep the bandwidth overhead low.

[13] This is already friendly to PTP+DTP+ε because to *guarantee* correct ε, $T_{recovery}$ should be the maximum recovery time, which is several seconds.

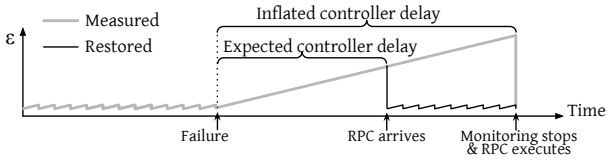


Figure 17: Restoration of ε under inflated controller delay.

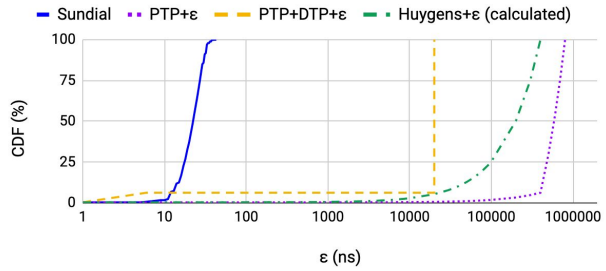


Figure 18: CDF of ε measured across devices without failures.

ered. To get the expected controller delay, we use its lower bound, the delay on the controller (without network delay), which is more friendly to schemes other than Sundial.

6.1.2 ε Distribution without Failures

Figure 18 shows the distribution of ε over all devices under different schemes. In Sundial, ε≤43ns, which matches the calculated value – the deepest device in the tree has ε_{base} of 25ns, and 90μs sync-interval leads to an additional 18ns.

In contrast, all other schemes have a much higher ε. In PTP+ε and PTP+DTP+ε, devices that do not directly synchronize to the root have to set T_{last_sync} earlier than T_{last_msg} by 2s and 100ms respectively, to account for possible failure-induced out-of-sync periods, so their ε can go up to 800μs and 20μs respectively during a sync-interval. Devices directly synchronizing to the root can set T_{last_sync} to T_{last_msg} and achieve lower ε. So their ε increases from 5ns (1-hop ε_{base}) to ∼400μs and 6ns respectively (2s and 5μs sync-intervals lead to 400μs and 1ns additional ε respectively at the end of each sync-interval). For these devices (∼6.3% of all), PTP+DTP+ε's low ε shows the benefit of extremely small sync-interval when failure is not a concern. Note that if available, Sundial can also benefit from DTP's physical layer design to futher reduce sync-interval. In Huygens+ε, during each 2s interval, ε increases from 0 to 400μs. Reducing sync-interval comes with CPU cost (Huygens already consumes 0.44% CPU of the whole cluster). However, even if the sync-interval was halved, ε is still 3 orders of magnitude higher than Sundial's.

6.1.3 ε Distribution during Failures

To understand the behavior under failures, we inject 50 random failures over a course of 6 minutes including 24 single link failures, 23 non-root single device failures, 2 domain failures and 1 root failure.

Figure 19 shows the time series of ε of a device affected by a link failure. In Sundial, ε is sawtooth between 15ns and 33ns during normal time, because this device has a depth of 3 in the tree.[14] When the link failure happens, ε increases

[14]Figure 21 shows the behavior at smaller timescales.

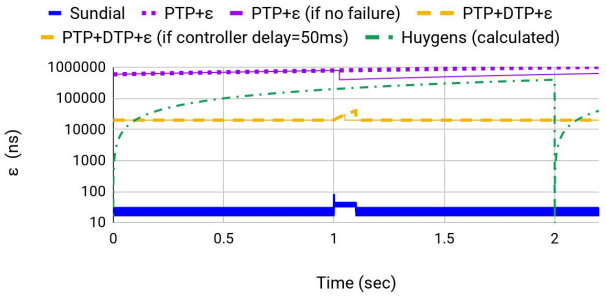


Figure 19: Time series of ε of a device affected by a link failure. The failure happens at 1s and the controller reacts to it near 1.1s.

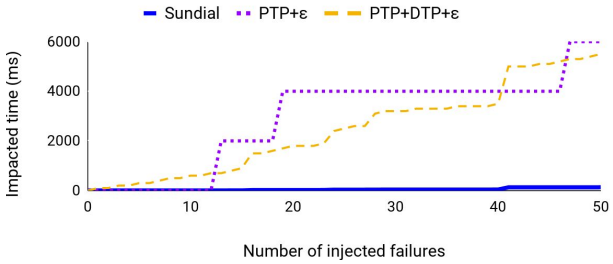


Figure 20: Blast radius of failures under different schemes. Impacted device time is the summation of per-device impacted time – duration when a device stops receiving sync-messages – over all devices .

to a maximum of 84ns and goes down in just 270μs (after the 185μs timeout, the next message is at 270μs). After that, ε is sawtooth between 30ns and 48ns, because its ε_{base} is set to $\varepsilon_{base,backup}$ by the local recovery, which is 30ns ($depth_{backup}$=6). Once the controller reconfigures the spanning tree, ε goes back to between 15ns and 33ns because its depth is 3. In PTP+ε, since the sync-message is dropped due to this failure, ε continues to increase for the next 2 seconds. Even if the sync-message was not dropped, ε for PTP+ε (w/o failure) remains high. PTP+DTP+ε's ε increases to 40μs and recovers to 20μs when the controller recovers the connectivity. However, even if the controller delay was lower (50 ms), it only reduces the peak ε to 30μs, but the normal ε is still around 20μs. Huygens+ε is not affected by failures, but its ε is normally very large (200μs at median and up to 400μs).

The behavior is similar under other failures – ε depends on the recovery time. For PTP+ε and PTP+DTP+ε, the recovery time depends on how long it takes for the controller to recover from it. For Sundial, the recovery time is much smaller as it's local. Any non-root failure recovery time is around 270μs, as is the case in Figure 19. The root failure takes slightly longer to recover from (365μs after the two timeouts) and ε increases to up to 103ns. The devices at different levels in the tree have slightly different ε (discussed in §6.1.4).

We now study the spatial and temporal impact range (blast radius) of failures. Figure 20 shows that Sundial's blast radius is very small. Even after 50 failures, the total impacted time summarized over all devices is only 131ms. The most significant jump happens when the root fails (40-th failure). PTP+ε and PTP+DTP+ε's blast radius is much higher owing to their longer recovery time. Note that more devices are affected by

failures under Sundial (401 in total) than under PTP+ε (3 in total) and PTP+DTP+ε (55 in total) as Sundial's backup-plan-based recovery can affect remote devices as well (those under the subtree of the failure). Even then the total impacted time for Sundial remains significantly smaller.

PTP+ε exhibits a step function because only failures occuring close to sync-interval boundaries affect it as the sync-interval of 2s is longer than the time to recover in most cases. The impact, however, is larger than in other schemes because it takes 2s for the next sync-message. PTP+DTP+ε's sync-interval is only 5μs and thus, every failure affects it. While Huygens+ε is not affected by connectivity failures, its ε remains high as shown before.

6.1.4 Microbenchmarks

How Sundial's different techniques improve ε. We zoom into details of how each technique improves ε. Specifically, starting with PTP+ε, we add (1) frequent sync-messages, (2) synchronous messaging, and (3) backup plan to it one by one, resulting in four schemes: PTP+ε, PTP+ε+freq_msg, PTP+ε+freq_msg+sync_msging, and Sundial itself.

Figure 21 shows the time series of ε during a link failure. Frequent sync-messages improve ε by an order of magnitude. Synchronous messaging further reduces ε during normal time as it helps each device detect connectivity failures: as long as a device receives a sync-message, it is connected to the root, so T_{last_sync} can be safely set to T_{last_msg}. Finally, adding the backup plan significantly speeds up the failure recovery – ε only increases for 270μs to a maximum of 84ns before the backup plan is activated, two orders of magnitude lower.

To show how Sundial's backup plan handles domain failures, we also run Sundial without considering domain failures (called Sundial w/o domain). We find that if a domain failure simultaneously takes down both the primary and backup parents of a device, the device's ε is like PTP+ε+freq_msg+sync_msging in Figure 21. This is expected because a down backup parent is equivalent to no backup parent. But if the failure domain is considered in the backup plan, ε is similar to Sundial in Figure 21, because the backup plan guarantees that no device loses both its primary and backup parents due to this domain failure. We also try another domain failure, which gradually takes down the primary and backup parents of a device, mimicing the domain failure that gradually takes down multiple devices or links (e.g., Figure 4). The result is similar.

Distribution of ε at different levels of the tree. We plot the maximum ε across devices at different depths, under different scenarios, shown in Figure 22. Root's ε is always 0. ε increases linearly with depth, which is expected as each level increments ε_{base} by 5ns.

6.2 Large-scale Simulations

We compare Sundial vs Marzullo's algorithm [27], an agreement algorithm for fault-tolerant clock-synchronization which is used by NTP [28] and TrueTime [12]. Marzullo's algorithm

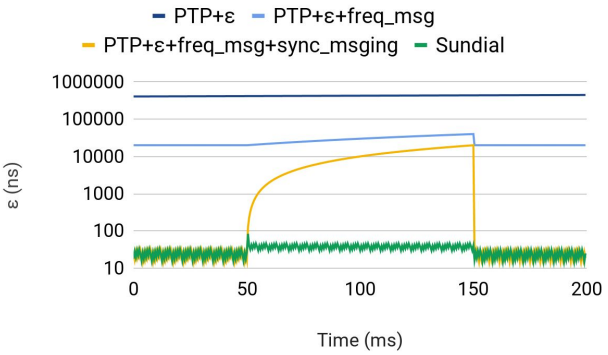


Figure 21: A link failure happens at 50 ms. The controller reacts to the failure at around 150 ms.

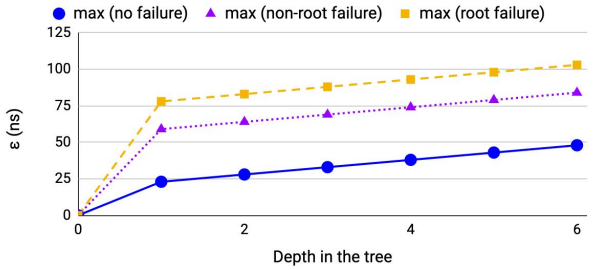


Figure 22: Distribution of ε at different levels in the tree.

also introduces time-uncertainty bound (ε) (called as error-bound in the original version). Since it is not supported in hardware due to its complexity, we use large scale simulations to demonstrate the performance characteristics.

Marzullo's algorithm synchronizes clocks through a mesh, so it can tolerate connectivity failures but has higher ε (§4.5). To reconcile the different time values and ε from multiple clocks, each node does intersection of time-uncertainty ranges of different clocks as the correct time should be within all ranges. A set of master clocks (1 or more clocks synchronized via GPS) serve as the source of synchronization, whose ε is always close to zero. Broken clocks can also be detected when the intersection result is empty. We simulate in a Jupiter topology [33] with 88,064 devices, where each node sends sync-messages to all its neighbors to maximize the tolerance to failures. We set 2 masters to tolerate master failures. The sync-interval is 90μs, same as Sundial. Figure 23 shows that during the normal time, Sundial has smaller ε than Marzullo's algorithm. Under failures, Marzullo's algorithm's ε is affected insignificantly. For Sundial, ε increases during failure recovery; the largest ε is 178ns, which is under the root failure.

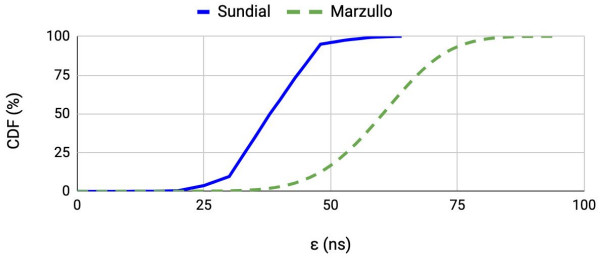


Figure 23: CDF of ε during normal time in Jupiter in simulation.

	DTP [21]	Huygens [13]	Marzullo [27]	PTP boundary clock [4]	Sundial
Message type	*Special PHY*	L3	Unspecified	L2	L2
Dealing with delay noises	*Neighbor*	Multi. msg	Unspecified	*Neighbor*	*Neighbor*
Synchronization structure	No master	Master, mesh	Master, mesh	Master, tree	*Master, mesh+tree*
Support time-uncertainty bound	No	No	*Yes*	No	*Yes*

Table 1: Design choices of state-of-the-art clock synchronization schemes. Italic options are the best.

6.3 Application Performance Improvement

Distributed transactional system. We evaluate the impact of smaller time-uncertainty bound using a load-test provided to us by Spanner team [12]. We run the load-test inside a datacenter. The load-test does 4KB transactions and we measure commit-wait gap – time to wait out time-uncertainty before committing the transaction. Results are in Table 2 where we show that our system improves performance by 3-4× not only in the median but also at the 99-th percentile.

	Baseline	With Sundial
Median	211μs	49μs
99-%ile	784μs	238μs

Table 2: Sundial improves commit-wait latency by 3-4× for Spanner running inside a datacenter.

Congestion Control. Delay-based congestion control such as Swift [17] is widely used in datacenters relying on end-to-end RTT measurements to control sending rate. A key challenge with such schemes is how to differentiate between forward and reverse-path congestion. As an example, congestion in the reverse path can also inflate RTT causing a sender to slow down even though there is no congestion in the forward path.[15] Synchronized clocks solve this problem as they enable the measurement of one-way delay (OWD) which can pinpoint the direction in which congestion is occurring.

We perform a microbenchmark with 3 servers – A, B and C with Swift congestion control. First, we only send traffic from A to B which achieves line-rate throughput. Next, we introduce reverse-path congestion by adding traffic from B and C to A. In Table 3, we observe A's throughput goes down to 50Gbps even though there was no congestion in the forward path. Replacing RTT with OWD as measured using Sundial resolves this completely and A continues to send at line rate.

	RTT	OWD
No reverse congestion	80.1 Gbps	80.5 Gbps
Reverse congestion	50.5 Gbps	80.9 Gbps

Table 3: Using one-way delay (OWD) improves throughput in the presence of reverse-path congestion.

7 Related Work

Other clock synchronization systems. Table 1 compares state-of-the-art solutions, in the design space outlined in §4.5.

DTP [21] introduces a specialized PHY layer to achieve zero bandwidth overhead of sync-messages. If this modified PHY can be standardized and productionized in the future, Sundial can readily benefit from it to have even lower sync-interval and ε. However, DTP does not reflect physical time since it doesn't have a master clock.

Huygens [13] does not synchronize switches, so it uses multiple messages between each pair to filter out noises. As a result, Huygens' sync-interval is limited, so it cannot achieve tight ε. Huygens' main advantage is that it is implemented completely in software and doesn't require hardware support (other than hardware timestamps) but it does not consider ε; and if incorporated, Huygens' ε is large primarily due to the large sync-interval. While it assumes clocks drift slowly during normal time, it cannot set a small *max_drift_rate* as the maximum drift is subject to failures (§3.2.1); otherwise it risks datacenter-wide application-level errors (e.g., inconsistent transactions), which is unacceptable.

Marzullo's algorithm [27] is the first to introduce ε but its ε is high because it sends messages through a mesh. PTP boundary clock [4] is based on a tree, and is not fault-tolerant.

Other solutions are too expensive (e.g., GPS [22]), too complex [18, 20, 31] or do not provide physical time [30, 34, 36].

Fault tolerance in other systems. In distributed systems and networking, fault tolerance is provided through redundancy [5, 10, 14, 19, 33, 38]. However, Sundial's backup plan cannot be chosen arbitrarily and needs to satisfy a set of properties (§4.2.1) to be generic to different types of failures.

Ethernet uses spanning tree protocols [2, 15] that can recompute a spanning tree in a distributed fashion after a failure, but they usually take up to a few seconds to converge [15].

8 Conclusion

Sundial is the first submicrosecond-level clock synchronization system that is resilient to failures. It uses hardware-software codesign to quickly detect failures and recover from them. Our evaluation shows that Sundial provides ∼100ns time-uncertainty bound under different types of failures, and improves performance in Spanner and in Swift.

9 Acknowledgements

We thank our shepherd Lorenzo Alvisi and OSDI reviewers for their helpful feedback. We also thank Arjun Singh, Jakov Seizovic, David Wetherall, David Dillow, Joe Zbiciak, and Xian Wu for the constructive feedback, Peter Cuy, Alex Iriza, and Bryant Chang for guidance on implementation, Shin Mao, Nanfang Li, and Ioannis Giannakopoulos for help on experimental evaluation.

[15] While prioritizing the ACK may solve the problem, it is impractical in production because of two reasons. (1) Network priorities are typically tied to business priorities; and we simply cannot send ACKs for lower business priority traffic on a higher network priority. (2) Sending ACKs on a higher network priority precludes ACK piggybacking on data packets, thereby increasing the packets-per-second to process. This is especially detrimental for CPU-efficient networking stacks such as PonyExpress in Snap [26].

References

[1] Broadcom: Timing over Packet (ToP) Processor for Precision Timing Applications. https://www.broadcom.com/products/embedded-and-networking-processors/communications/bcm53903.

[2] IEEE 802.1D Work Group, IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges, 2004.

[3] CockroachDB, 2008. https://github.com/cockroachdb/.

[4] IEEE Standard 1588-2008, 2008. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757.

[5] Redundancy N+1, N+2 vs. 2N vs. 2N+1, 2014. https://www.datacenters.com/news/redundancy-n-1-n-2-vs-2n-vs-2n-1.

[6] IEEE 1588 PTP and Analytics on the Cisco Nexus 3548 Switch, 2017. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white-paper-c11-731501.html.

[7] Clock Oscillators Surface Mount Type KC3225L-P2/ KC3225L-P3 Series, 2018. https://global.kyocera.com/prdct/electro/pdf/kc3225l_p2p3_e.pdf.

[8] Juniper Precision Time Protocol Overview, 2020. https://www.juniper.net/documentation/en_US/junos/topics/concept/ptp-overview.html.

[9] Mellanox Precision Time Protocol, 2020. https://docs.mellanox.com/display/ONYXv381174/Precision+Time+Protocol.

[10] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 2008.

[11] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 1985.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[13] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, 2018.

[14] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM*, 2009.

[15] New BPDU Handling. Understanding rapid spanning tree protocol (802.1 w). *Catalyst*, 2948(L3/4908G):L3.

[16] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, 1987.

[17] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the ACM SIGCOMM*, 2020.

[18] Leslie Lamport. *Synchronizing time servers*. Digital, Systems Research Center, 1987.

[19] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[20] Leslie Lamport and Peter M Melliar-Smith. Byzantine clock synchronization. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 68–74, 1984.

[21] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[22] Wlodzimierz Lewandowski, Jacques Azoubib, and William J Klepczynski. Gps: Primary tool for time transfer. *Proceedings of the IEEE*, 87(1):163–172, 1999.

[23] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[24] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center

networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, 2016.

[25] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[26] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[27] Keith Marzullo and Susan Owicki. Maintaining the time in a distributed system. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 295–305, 1983.

[28] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

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

[30] Luca Schenato and Federico Fiorentin. Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks. *Automatica*, 47(9):1878–1886, 2011.

[31] Ulrich Schmid. Synchronized utc for distributed real-time systems. *Annual Review in Automatic Programming*, 18:101–107, 1994.

[32] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojeviundefined, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. SIGMOD '19, 2019.

[33] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 2015.

[34] Roberto Solis, Vivek S Borkar, and PR Kumar. A new distributed time synchronization protocol for multihop wireless networks. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 2734–2739. IEEE, 2006.

[35] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[36] Geoffrey Werner-Allen, Geetika Tewari, Ankit Patel, Matt Welsh, and Radhika Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, 2005.

[37] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[38] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding lifecycle management complexity of datacenter topologies. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[39] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[40] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.



Fault-tolerant and Transactional Stateful Serverless Workflows

Haoran Zhang, Adney Cardoza†, Peter Baile Chen, Sebastian Angel, and Vincent Liu
University of Pennsylvania †*Rutgers University-Camden*

Abstract

This paper introduces Beldi, a library and runtime system for writing and composing fault-tolerant and transactional stateful serverless functions. Beldi runs on existing providers and lets developers write complex stateful applications that require fault tolerance and transactional semantics without the need to deal with tasks such as load balancing or maintaining virtual machines. Beldi's contributions include extending the log-based fault-tolerant approach in Olive (OSDI 2016) with new data structures, transaction protocols, function invocations, and garbage collection. They also include adapting the resulting framework to work over a federated environment where each serverless function has sovereignty over its own data. We implement three applications on Beldi, including a movie review service, a travel reservation system, and a social media site. Our evaluation on 1,000 AWS Lambdas shows that Beldi's approach is effective and affordable.

1 Introduction

Serverless computing is changing the way in which we structure and deploy computations in Internet-scale systems. Enabled by platforms like AWS Lambda [2], Azure Functions [3], and Google Cloud Functions [18], programmers can break their application into small functions that providers then automatically distribute over their data centers. When a user issues a request to a serverless-based system, this request flows through the corresponding functions to achieve the desired end-to-end semantics. For example, in an e-commerce site, a user's purchase might trigger a product order, a shipping event, a credit card charge, and an inventory update, all of which could be handled by separate serverless functions.

During development, structuring an application as a set of serverless functions brings forth the benefits of microservice architectures: it promotes modular design, quick iteration, and code reuse. During deployment, it frees programmers from the prosaic but difficult tasks associated with provisioning, scaling, and maintaining the underlying computational, storage, and network resources of the system. In particular, app developers need not worry about setting up virtual machines or containers, starting or winding down instances to accommodate demand, or routing user requests to the right set of functional units—all of this is automated once an app developer describes the connectivity of the units.

A key challenge in increasing the general applicability of serverless computing lies in correctly and efficiently composing different functions to obtain nontrivial end-to-end applications. This is fairly straightforward when functions are stateless, but becomes involved when the functions maintain their own state (e.g., modify a data structure that persists across invocations). Composing such *stateful serverless functions* (SSFs) requires reasoning about consistency and isolation semantics in the presence of concurrent requests and dealing with component failures. While these requirements are common in distributed systems and are addressed by existing proposals [8, 28, 33, 35, 46], SSFs have unique idiosyncrasies that make existing approaches a poor fit.

The first peculiarity is that request routing is stateless. Approaches based on state machine replication are hard to implement because a follow-up message might be routed by the infrastructure to a different SSF instance from the one that processed a prior message (e.g., an "accept" routed differently than its "prepare"). A second characteristic is that SSFs can be independent and have sovereignty over their own data. For example, different organizations may develop and deploy SSFs, and an application may stitch them together to achieve some end-to-end functionality. As a result, there is no component in the system that has full visibility (or access) to all the state. Lastly, SSF workflows (directed graphs of SSFs) can be complex and include cycles to express recursion and loops over SSFs. If a developer wishes to define transactions over such workflows (or its subgraphs), *all* transactions (including those that will abort) must observe consistent state to avoid infinite loops and undefined behavior. This is a common requirement in transactional memory systems [20, 23, 32, 37, 38], but is seldom needed in distributed transaction protocols

To bring fault-tolerance and transactions to this challenging environment, this paper introduces *Beldi*, a library and runtime system for building workflows of SSFs. Beldi runs on existing cloud providers without any modification to their infrastructure and without the need for servers. The SSFs used in Beldi can come from either the app developer, other developers in the same organization, third-party open-source developers, or the cloud providers. Regardless, Beldi helps to stitch together these components in a way that insulates the developer from the details of concurrency control, fault tolerance, and SSF composition.

A well-known aspect of SSFs is that even though they can persist state, this state is usually kept in low-latency NoSQL databases (possibly different for each SSF) such as DynamoDB, Bigtable, and Cosmos DB that are already fault tolerant. Viewed in this light, SSFs are clients of scalable fault-tolerant storage services rather than stateful services themselves. Beldi's goal is therefore to guarantee *exactly-once* semantics to workflows in the presence of clients (SSFs) that fail at any point in their execution and to offer *synchro-*

nization primitives (in the form of locks and transactions) to prevent concurrent clients from unsafely handling state.

To realize this vision, Beldi extends Olive [36] and adapts its mechanisms to the SSF setting. Olive is a recent framework that exposes an elegant abstraction based on logging and request re-execution to clients of cloud storage systems; operations that use Olive's abstraction enjoy exactly-once semantics. Beldi's extensions include support for operations beyond storage accesses such as synchronous and asynchronous invocations (so that SSFs can invoke each other), a new data structure for unifying storage of application state and logs, and protocols that operate efficiently on this data structure (§4). The purpose of Beldi's extensions is to smooth out the differences between Olive's original use case and ours. As one example, Olive's most critical optimization assumes that clients can store a large number of log entries in a database's *atomicity scope* (the scope at which the database can atomically update objects). However, this assumption does not hold for many databases commonly used by SSFs. In DynamoDB, for example, the atomicity scope is a single row that can store at most 400 KB of data [14]—the row would be full in less than a minute in our applications.

Beldi also adapts existing concurrency control and distributed commit protocols to support transactions over SSF workflows. A salient aspect of our setting is that there is no entity that can serve as a coordinator: a user issues its request to the first SSF in the workflow, and SSFs interact only with the SSFs in their outgoing edges in the workflow. Consequently, we design a protocol where SSFs work together (while respecting the workflow's underlying communication pattern) to fulfill the duties of the coordinator and collectively decide whether to commit or abort a transaction (§6).

To showcase the costs and the benefits of Beldi, we implement three applications as representative case studies: (1) a travel reservation system, (2) a social media site, and (3) a movie review service. These applications are based on DeathStarBench [12, 16], which is an open-source benchmark for microservices; we have ported and extended these applications to work without servers using SSFs. Our evaluation on AWS reveals that, at saturation, Beldi's guarantees come at an increase in the median request completion time of 2.4–3.3×, and 99th percentile completion time of 1.2–1.8× (§7.4). At low load, the median completion time increase is under 2×.

In summary, Beldi helps developers build fault-tolerant and transactional applications on top of SSFs at a modest cost. In doing so, Beldi simplifies reasoning about compositions of SSFs, runs on existing serverless platforms without modifications, and extends an elegant fault-tolerant abstraction.

2 Background and Goals

In this section, we describe the basics of serverless computing (sometimes known as Function-as-a-Service), the challenge of deploying complex serverless applications that incorporate state, and a list of requirements that Beldi aims to satisfy.

2.1 Serverless functions

Serverless computing aims to eliminate the need to manage machines, runtimes, and resources (i.e., everything except the app logic). It provides an abstraction where developers upload a simple function (or 'lambda') to the cloud provider that is invoked on demand; an identifier is provided with which clients and other services can invoke the function.

The cloud provider is then responsible for provisioning the VMs or containers, deploying the user code, and scaling the allocated resources up and down based on current demand—all of this is transparent to users. In practice, this means that on every function invocation the provider will spawn a new *worker* (VM or container) with the necessary runtime and dispatch the request to this worker ('cold start'). The provider may also use an existing worker, if one is free ('warm start'). Note that while workers can stay warm for a while, running functions are limited by a timeout, after which they are killed. This time limit is configurable (up to 15 min in 1 s increments on AWS, up to 9 min in 1 ms increments on Google Cloud, and unbounded time in 1 s increments on Azure) and helps in budgeting and limiting the effect of bugs.

Serverless functions are often used individually, but they can also be composed into *workflows*: directed graphs of functions that may contain cycles to express recursion or loops over one or more functions. Some ways to create workflows include AWS's *step functions* [41] and *driver functions*. A step function specifies how to stitch together different functions (represented by their identifiers) and their inputs and outputs; the step function takes care of all scheduling and data movement, and users get an identifier to invoke it. In contrast, a driver function is a single function specified by the developer that invokes other functions (similar to the main function of a traditional program). Control flow can form a graph because functions (including the driver function) can be multi-threaded or perform asynchronous invocations.

Stateful serverless functions (SSFs). Serverless functions were originally designed to be stateless. As such, state is not guaranteed to persist between function invocations—even when writing to a worker's local disk, the function's context can be terminated as part of dynamic resource management, or load balancing might direct follow-up requests to different or new instances. Accordingly, a common workaround to persist data is to store it in fault-tolerant low-latency NoSQL databases. For example, AWS Lambdas can persist their state in DynamoDB, Google cloud functions can use Cloud Bigtable, and Azure functions can use Cosmos DB. Through these intermediaries, *stateful serverless functions* (SSFs) can save state and expose it to other instances.

Unfortunately, the above approach to state interacts poorly with the way that serverless platforms handle failures. If a function in a workflow crashes or its worker hangs, the provider will either (1) do nothing, leaving the workflow incomplete, or (2) restart the function on a different worker,

potentially incrementing a counter twice, popping a queue multiple times, or corrupting database state and violating application semantics. Indeed, serverless providers currently recommend that developers write SSFs that are idempotent to ensure that re-execution is safe [17]. While helpful, these recommendations place the burden entirely on developers. In contrast, Beldi simplifies this process so developers need only worry about their application logic and not the low-level details of how serverless providers respond to failures.

2.2 Requirements and assumptions

We strive to design a framework that helps developers build serverless applications that tolerate failures and handle concurrent operations correctly. Our concrete goals are:

Exactly-once semantics: Beldi should guarantee *exactly-once* execution semantics in the presence of SSF or worker crash failures. That is, even if an SSF crashes in the midst of its execution and is restarted by the provider an arbitrary number of times, the resulting state must be equivalent to that produced by an execution in which the SSF ran exactly once, from start to finish, without crashing.

SSF data sovereignty: Beldi should support SSFs that are developed and managed independently. For example, multiple instances of an SSF may all access the same database, but they might not have access to the databases of other SSFs, even those in the same workflow. Instead, state should only be exposed by choice through an SSF's outputs. This type of encapsulation is important to support a paradigm in which different developers, organizations, and teams within the same organization are responsible for designing and maintaining their own SSFs. An application developer can then contract with SSF developers (or teams) to integrate their SSFs into the application's workflow via the SSF's identifier (§2). Furthermore, data sovereignty is key to enabling developers to offer proprietary functions-as-a-service to others, and is a best practice in microservice architectures [11, §4]. For example, Microsoft's eShopOnContainers [29] serves as a blueprint for applying these ideas to real-world applications.

SSF reusability: Beldi should allow multiple applications to use the same SSFs in their workflows at the same time. This may require each SSF to have different tables or databases to maintain the state of each application separately, though cross-application state should also be supported.

Workflow transactions: Beldi should support an optional transactional API that allows an application to specify any subgraph of a workflow that should be processed transactionally with ACID semantics. We target *opacity* [20] as the isolation level. Opacity ensures that (1) the effects of concurrent transactions are equivalent to some serial execution, and (2) every transaction, including those that are aborted, always observes a consistent view of the database. We discuss why these requirements are important in SSFs in Section 6.2.

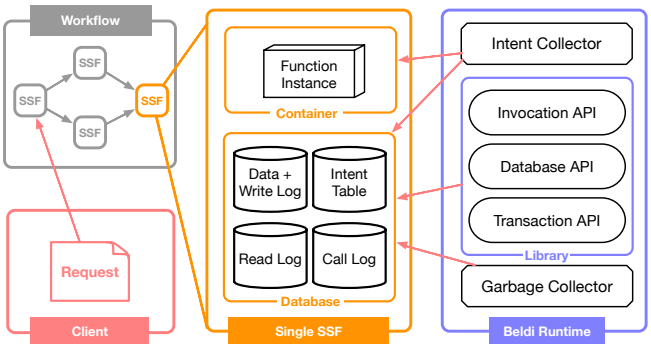


FIGURE 1—Beldi's architecture. Developers write SSFs as they do today, but use the Beldi API for transactions and externally visible operations. At runtime, operations for each SSF are logged to a database, which, when combined with a per-SSF intent and garbage collector, guarantees exactly-once semantics.

Deployable today: Beldi should work on existing serverless platforms without any modifications on their end. This allows developers to use Beldi on any provider of their choosing (or even a multi-provider setup), and lowers the barrier to switch providers. Additionally, developers should not need to run any servers in order to use Beldi. After all, a big appeal of serverless is that it frees developers from such burdens.

Assumptions. To achieve these goals, Beldi makes some assumptions about the storage provided to SSFs: that it supports strong consistency, tolerates faults, supports atomic updates on some atomicity scope (e.g., row, partition), and has a scan operation with the ability to filter results and create projections. These assumptions hold for the NoSQL databases commonly used by SSFs: Amazon's DynamoDB, Azure's Cosmos DB, and Google's Bigtable.

3 Design Overview

Beldi consists of a library that developers use to write their SSFs and a serverless-based runtime system to support them. Beldi's approach to handling SSF failures is based on an idea most recently explored by Olive [36] and inspired by decades of work on log-based fault tolerance [19, 30]. Specifically, Beldi executes SSF operations while atomically logging these actions and periodically re-executes SSFs that have not yet finished. The logs prevent duplicating work that has already been done, guaranteeing *at-most-once* execution semantics, while the re-execution ensures *at-least-once* semantics.

Figure 1 depicts Beldi's high-level architecture. Beldi consists of four components: (1) the Beldi library, which exposes APIs for invocations, database reads/writes, and transactions; (2) a set of database tables that store the SSF's state, as well as logs of reads, writes, and invocations; (3) an *intent collector*, which is a serverless function that restarts any instances of the corresponding SSF that have stalled or crashed; and (4) a *garbage collector*, which is a serverless function that keeps the logs from growing unboundedly.

To ensure data sovereignty (§2.2), the runtimes and logs

of different SSFs are independently managed and stored; however, all instances of *related* SSFs may share the same Beldi infrastructure. An app developer composes multiple SSFs into a workflow by chaining them together using a driver function, a step function, or a combination of the two. In the following sections we expand on each of these components.

3.1 Initial inspiration: Olive

Olive [36] guarantees exactly-once execution semantics for clients that may fail while interacting with a fault-tolerant storage server. This is a similar objective as ours, though our setting makes applying Olive's ideas nontrivial. An intent in Olive is an arbitrary code snippet that the client intends to execute with exactly-once semantics. Each intent is assigned a unique identifier (*intent id*), which Olive uses as the primary key to save its progress. A client in Olive enjoys *at-most-once* semantics by checking the intent's progress and skipping completed operations during re-execution. Intents consist of local and external operations. For example, incrementing a local variable is a local operation, whereas reading a value from storage is an external operation. Each external operation in the intent is assigned a monotonically increasing *step number*, starting at 0, that uniquely identifies it.

There are two key requirements for intents. First, intents must be deterministic; developers can make non-deterministic operations (e.g., a call to a random number generator) deterministic by logging their results and replaying the same values in the event of a re-execution. Second, intents must be guaranteed to always complete in the absence of failures (e.g., they must be free from bugs such as deadlock or infinite loops).

After an intent has been successfully logged, the client in Olive executes the intent's code normally until it reaches an external operation (e.g., reading or writing to the database). Then, the client: (1) determines the operation's step number; (2) performs the operation (e.g., writes to the database); (3) logs the intent id, step number, and the operation's return value (if any) into a separate database table called the *operation log*. When the client completes all operations, it marks the intent as 'done' in the intent table.

To ensure at-most-once execution semantics, the client in Olive must perform actions (2) and (3) above atomically. This ensures that if Olive re-executes an intent, there will be a record in the operation log showing that a particular step has already been completed and should not be re-executed. Instead, the entity re-executing the intent should resume execution from the last completed step, using logged return values from previous steps as needed. To make these two actions atomic, Olive introduces a technique called *Distributed Atomic Affinity Logging* (DAAL), which collocates log entries for an item in the same *atomicity scope* (the scope at which the database supports atomic operations) with the item's data. For example, in a storage system where operations are atomic at the row level, Olive would store the item's value and its log entries in different columns of the same row.

Beldi Library Function	Description
`read(k)` → `v`	Read operation
`write(k, v)`	Write operation
`condWrite(k, v, c)` → T/F	Write if `c` is true
`syncInvoke(s, params)` → `v`	Calls `s` and waits for answer
`asyncInvoke(s, params)`	Calls `s` without waiting
`lock()`	Acquire a lock
`unlock()`	Release a lock
`begin_tx()`	Begin a transaction
`end_tx()`	End a transaction

FIGURE 2—Beldi's API for SSFs, which includes all of Beldi's primitives and its transactional support (§6).

Intent collector. To guarantee *at-least-once* semantics, Olive must ensure that some entity finishes the intent if the client crashes. This is the job of the *intent collector* (IC), which is a background process that periodically scans the intent table and completes unfinished intents by running their code. Before the IC executes an external operation, it consults the operation log table with the operation's step number to see if the operation has already been done and to retrieve any return value; regular clients also perform this check between actions (1) and (2). If the operation has not been done, the IC atomically executes the operation and logs the result to the operation log table. Even if multiple IC instances execute concurrently, or if the IC starts executing the intent of a client that has not crashed, this is safe because of Olive's assurance of at-most-once semantics.

Beldi vs. Olive. Beldi is inspired by Olive's high-level approach but makes key changes and introduces new data structures and tables, support for invocations so that SSFs can call each other (Olive only supports storage operations), and garbage collection mechanisms to keep overheads low.

An important difference between the two is the definition of an 'intent.' In Olive, intents are code snippets—logged by the client—and all intents are logged in the same intent table. In Beldi, the *client* (which is the SSF) is the code snippet. As a result, an intent in Beldi is not code but rather the parameters that identify a particular running instance of the SSF: its inputs, start time, whether it was launched asynchronously, etc. Accordingly, Beldi uses the term 'instance id' instead of 'intent id' to capture this distinction.

Another critical difference is that, as shown in Figure 1, each SSF in Beldi is backed by a different database and Beldi runtime to ensure data sovereignty, though different SSFs developed by the same engineering team may reuse these components if desired. We will expand on these details in the following sections, but we begin by introducing Beldi's API.

3.2 Beldi's API

Beldi exposes the API in Figure 2, which includes key-value operations such as `read`, `write`, and `condWrite` (a write that succeeds only if the provided condition evaluates to true), and functions to invoke other SSFs (`syncInvoke` and

Log	Key	Value
`intent`	instance id	done, async, args, ret, ts
`read`	instance id, step number	value
`write`	instance id, step number	true / false
`invoke`	instance id, step number	instance id of callee, result

FIGURE 3—Beldi maintains four logs for each SSF. The intent table keeps track of an instance's completion status, arguments, return value, type of invocation, and timestamp assigned by its garbage collector (ts). The read log stores the value read. The write log stores true for writes, or the condition evaluation for a conditional write. The invoke log stores the instance id of the callee and its result.

`asyncInvoke`). These operations are meant as drop-in replacements for the existing interface used by SSFs. Furthermore, Beldi supports the ability to `begin` and `end` transactions; operations between these calls enjoy ACID semantics.

Beldi's API hides from developers all of the complexity of logging, replaying, and concurrency control protocols that take place under the hood to guarantee exactly-once semantics and support transactions. For example, an SSF using Beldi's API automatically determines (from the input, environment, and global variables) the SSF's instance id, step number, and whether it is part of a transaction. Beldi takes actions before and after the main body of the SSF as well as around any Beldi API operations.

3.3 Beldi's runtime infrastructure

Developers write SSF code as they do today, but link Beldi's library and use its API. The rest of Beldi's mechanisms happen behind the scenes.

Intent table. Beldi associates with every SSF invocation an *instance id*, which uniquely identifies an intent to execute a given SSF. For the first SSF in a workflow, the instance id is the UUID assigned by the serverless platform to the initial request. For example, in AWS this UUID is called the 'request id,' in GCP it is called the 'event id,' and in Azure it is the 'invocation id.' For subsequent SSFs in the workflow, each caller in the graph will generate a new UUID to be used by the callee as its instance id. A new id is generated even if the SSF has been invoked earlier in the workflow or if the callee is another instance of the caller SSF (in the case of recursive functions). Thus, every SSF instance will have a distinct instance id, even if the instances are of the same SSF and in the same workflow.

Beldi keeps an intent table that contains the instance id, arguments, completion status, and other information listed in Figure 3 for every SSF instance that users and other SSFs intend to execute. It does this by modifying SSFs to ensure that the first operation is to check the intent table to see if their instance id is already present and, if not, to log a new entry. Beldi performs a similar modification to set the intent as 'done' at the end of the SSF execution.

Operation logs. In addition to the intent table, Beldi maintains three logs for each SSF: a *read log*, *write log*, and *invoke log*. Their schema is also in Figure 3. For each operation, the key into the log is the combination of the executing SSF's instance id and the step number, which (like in Olive) is a counter that identifies each unique external operation. Each read operation adds the value read from the database into the read log. Writes, meanwhile, write to the write log with a boolean flag that states whether the write operation took effect. Regular writes always set this flag to true, while conditional writes set it to the outcome of the condition at the time of the write. The actual data being written is stored in a data table, although in Section 4 we discuss a data structure that generalizes Olive's DAAL and collocates the write log in the same table as the data to avoid cross-table transactions. The invoke log is new to Beldi and ensures at-most-once semantics for calls to other SSFs; we describe it in Section 4.5.

Intent and Garbage Collectors. For each SSF, Beldi introduces a pair of serverless functions that are triggered periodically by a timer. The first function acts as the SSF's *intent collector* (IC). The IC scans the SSF's intent table to discover instances of the SSF that have not yet finished (lack the 'done' flag). The IC restarts each unfinished SSF by re-executing it with the original instance id and arguments. Note that it is safe for the IC to restart an SSF instance even if the original instance is still running and has not crashed, owing to Beldi's use of logs to guarantee at-most-once semantics for each step of the SSF. We implement two natural optimizations for the IC. First, the IC restarts instances only after some amount of time has passed since the last time they were launched to avoid spawning too many duplicate instances in cases where the IC runs very frequently. Second, the IC speeds up the process of finding unfinished instances among all instance ids in the intent table by maintaining a secondary index.

The second function acts as a *Garbage Collector* (GC) for completed intents, taking care to ensure safety in the presence of concurrent SSF instances, IC instances, and even GC instances. This component is described in Section 5.

4 Executing and Logging Operations in Beldi

As we mention in Section 3.1, guaranteeing exactly-once semantics requires atomically logging and executing operations. This section discusses how Beldi achieves this.

4.1 Linked DAAL

The logging approach taken by Olive (§3.1) requires an *atomicity scope* with high storage capacity, as otherwise few log entries can be added. In the context of Cosmos DB (the successor of the database used by Olive), the atomicity scope is a database partition, and the atomic operation is a transactional batch update. Olive's DAAL is a good fit for Cosmos DB because partitions can hold up to 20 GB of data [10], which is enough to collocate a data item and a large number of log entries. However, other databases adopt designs with more limited atomicity scopes. For example, the atomicity scope of DynamoDB and Bigtable is one row, which can hold up to

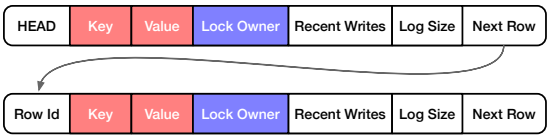


FIGURE 4—Linked DAAL for a single item. Each row contains the item's key, previous values (except the last row which contains the current value), lock information (used for transactions), a log of recent writes, and information for traversal and garbage collection.

400 KB [14] and 256 MB [7], respectively; the recommended limits are much lower. If we were to use Olive's DAAL with DynamoDB, an SSF could only perform hundreds of writes to a given key before filling up the row. At such point, Olive would be unable to make further progress until the logs are pruned. This is hard to do in our setting: reaching a state of quiescence where it is safe to garbage collect logs is challenging since existing platforms expose no mechanism to kill or pause SSFs (§5).

To support all common databases, Beldi introduces a new data structure called the *linked DAAL* that allows logs to exist on multiple rows (or atomicity scopes), with new rows being added as needed. There are three reasons why this simple data structure is interesting for our purposes: (1) linked DAALs continue to avoid the overheads of cross-table transactions and work on databases that do not support such transactions; (2) linked DAALs are a type of non-blocking linked list [21, 42, 47], allowing multiple SSFs to access them concurrently with the operations supported by the atomicity scope (e.g., atomic partial updates); (3) even with frequent accesses, our garbage collection protocol can ensure that the length of the list for each item is kept consistently small (§5).

Structure. Figure 4 gives an example of a linked DAAL for an item with two rows of logs. Every row stores the item's key, value, owner of the lock (used for transactions in Section 6), the log of writes, and metadata needed to traverse the linked DAAL and perform garbage collection. The first row is the 'head,' which has a special `RowId` and is never garbage collected. The primary key for rows is `RowId` + `Key`, the hash key is `Key`, and the sort key is `RowId`. When a row is full and the SSF issues a write operation, a new row is appended with the updated value and a log entry describing the write; the previous row's value and logs are not modified once filled. Thus, the tail always has the most recent value.

Traversal. Most operations in Beldi require traversal to the tail of the list. The simplest way to accomplish this is to start at the designated head row and iteratively issue read requests for each `NextRow` until the field is empty. While this procedure will eventually reach the tail, the number of database operations grows with the length of the list. Garbage collection can control this length, but Beldi applies an additional optimization that leverages the scan and projection operations available in the three NoSQL databases that we surveyed. Specifically, Beldi issues a single scan operation to

```
def read(table, key):
    linkedDAAL = rawScan(table,
        cond: "Key is {key}",
        project: ["RowId", "NextRow"])
    tail = getTail(linkedDAAL)
    val = rawRead(table, tail)
    logKey = [ID, STEP]
    STEP = STEP + 1
    ok = rawCondWrite(ReadLog, logKey,
        cond: "{logKey} does not exist"
        update: "Value = {val}")
    if ok:
        return val
    else:
        return rawRead(ReadLog, logKey)
```

FIGURE 5—Pseudocode for Beldi's read wrapper function. Functions beginning with "`raw`" refer to native (unwrapped) access to the database tables storing the data or the logs. Identifiers starting with capital letters indicate a member of the log structures.

the database that returns every row containing a target `Key`. On its own, the scan operation returns all contents of each row (including the values, write logs, etc.). To reduce this overhead, Beldi applies a projection that filters out all columns except for `RowId` and `NextRow`. This combination of scan and projection allows Beldi to download only 256 bits per row of the linked DAAL. From these rows, Beldi constructs a skeleton version of the linked DAAL locally, which it can quickly traverse to find the `RowId` of the tail.

We note that the individual reads in a scan are not executed atomically. For example, Beldi might see a row with no `NextRow`, and also receive a row that was subsequently appended to it. This operation might even retrieve rows that are orphaned from a failed append operation. Regardless, when these databases are configured to be linearizable [6, 9, 13], the set of rows traversed from the head to the first instance of an empty `NextRow` form a consistent snapshot of the linked DAAL—any write that completes strictly before the scan begins will be reflected in the constructed local linked DAAL.

While the linked DAAL is structurally simple, operating on it requires care. The following sections detail how Beldi's API functions read and modify the linked DAAL.

4.2 Read

We begin by discussing Beldi's `read` operation. While `read` has no externally visible effects on its own, the potential use of its non-deterministic results in a subsequent external operation means that Beldi must record the result of every `read` in a dedicated *ReadLog*. Unlike write operations, however, the read from the database and the log to the ReadLog need not happen atomically—if the SSF crashes before logging the outcome, it is fine to fetch a fresh value as the previous result did not have any externally visible effect.

Figure 5 shows the pseudocode of the `read` API function, which involves two steps: (1) read the most recent value of the key from the tail of the linked DAAL, and (2) log the result

```
def write(table, key, val):
    logKey = [ID, STEP]
    linkedDAAL = rawScan(table,
        cond: "Key is {key}"
        project: ["RowId", "NextRow",
                  "RecentWrites[{logKey}]"])
    if logKey not in linkedDAAL:
        tail = getTail(linkedDAAL)
        tryWrite(table, key, val, tail)
    STEP = STEP + 1
def tryWrite(table, key, val, row):
    logKey = [ID, STEP]
    ok = rawCondWrite(table, row[RowId],
        cond: "({logKey} not in RecentWrites)
               && (LogSize < N)",
        update: "Value = {val};
                 LogSize = LogSize + 1;
                 RecentWrites[{logKey}] = NULL")
    if ok: # Case B
        return
    row = rawRead(table, row[RowId])
    if logKey in row[RecentWrites]: # Case A
        return
    elif row[NextRow] does not exist: # Case D
        row = appendRow(table, key, row)
    else: # Case C
        row = rawRead(table, row[NextRow])
    tryWrite(table, key, val, row)
```

FIGURE 6—Pseudocode for Beldi's write wrapper function.

to the ReadLog if it has not yet been completed. For the first step, Beldi retrieves the tail as described in Section 4.1. For the second step, Beldi uses an atomic conditional update to efficiently log the operation without overwriting a previously executed read. If it encounters a conflict during the update, it returns the previous result from the ReadLog.

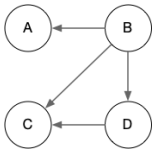
4.3 Write

A write is more complex as the update and logging must be done atomically—within the same atomicity scope—and Beldi needs to handle cases where other SSFs are accessing and appending to the linked DAAL concurrently. At a high level, the write operation must find the tail of the linked DAAL, check if the write has been previously executed, log/update the tail if it has not, and extend the linked DAAL if the current tail is full. Like `read`, Beldi can use scan and projection to assemble a minimal local version of the linked DAAL. Unlike `read`, Beldi cannot skip directly to the tail; instead, Beldi must check that none of the scanned rows contains a record of the current operation. Furthermore, once Beldi has a candidate for the tail, Beldi needs to update its value and add an entry to its log atomically. For a given tail candidate there are exactly four possible scenarios:

A. The operation has already been executed and the [instance ID, step number] tuple can be found in the current row. Beldi can return immediately in this case.
B. The operation is not in the log and there is still space. This indicates that Beldi is at the tail, the operation has never been executed previously, and there is room in the current row to execute/log the write.
C. The operation is not in the log, but the log is full and there is a pointer to the next row. Beldi should follow the provided pointer toward the tail.
D. The operation is not in the log and the log is full, but there is no next row. Beldi should append a new row and advance to that new row.

	logKey ∈ logs	logSize < N	∃ nextRow
A	True	*	*
B	False	True	False
C	False	False	True
D	False	False	False

(a) Cases



(b) Transitions

FIGURE 7—Possible cases for the state of a candidate tail in the linked DAAL during a `write` and its potential transitions.

We formulate a lock-free algorithm to handle all the cases above by examining the transitions induced by concurrent SSF accesses. For example, if Beldi is in case B, where the operation is not in any log and there is still space to execute it in the current row, a concurrent SSF can, without warning, execute the current operation (→A) or fill the remaining space in the log (→C/D). The reverse is not true: once there is a `NextRow` pointer, the linked DAAL will never revert to having extra space for logs. The cases and their transitions are summarized in Figure 7, where N is the maximum number of log entries that can fit in a row when accounting for the size of the key, value, and other metadata. The exception is garbage collection (not covered in Figure 7), whose operation and correctness we describe in Section 5. An arrow in Figure 7b indicates a possible effect of concurrent SSF instances.

To safely identify the state of a row, Beldi checks for each case starting at the node(s) in the transition graph without incoming edges. In this case there is only one such node (B), so Beldi performs a conditional write with the condition given in case B of Figure 7a (i.e., that the logKey is not in the logs, that the logSize is less than N, and that there is no nextRow). If the conditional check fails, the state of the row will not revert back to case B later because B has no incoming edges. Therefore, it is safe to remove B from the transition graph and check the remaining cases. Beldi repeats the above process with cases A and D (in any order) because they they have no incoming edges in the remaining graph. Finally, if all prior conditions fail, the row is in case C.

4.4 Conditional write

Beldi also provides support for conditional writes, which only execute if a user-defined condition is true at the time of the write. The initial scan and subsequent scenarios are similar to the scenarios for unconditional writes. The only exception is the case where the operation has not previously executed and the current row still has remaining space in the log (i.e., case B from Section 4.3). We split this case into two: in B_1, the condition is true, and in B_2, the condition is false.

```
def syncInvoke(callee, input):
    calleeId = UUID()
    logKey = [ID, STEP]
    STEP = STEP + 1
    ok = rawCondWrite(InvokeLog, logKey,
        cond: "{logKey} not in InvokeLog"
        update: "Id = {calleeId};
                 Result = NULL")
    if not ok:
        record = rawRead(InvokeLog, logKey)
        calleeId = record[Id]
        result = record[Result]
    if result does not exist:
        return rawSyncInvoke(callee,
            [calleeId, input])
# When the Callee is done it issues a callback
# to the caller. Below is the callback handler.
def syncInvokeCallbackHandler(calleeId, result):
    rawWrite(InvokeLog, cond: "Id = {calleeId}",
             update: "Result = {result}")
```

FIGURE 8—Pseudocode for synchronous invocation of other SSFs. Asynchronous invocations are similar, but since they do not have return values, the callback is invoked as soon as the callee logs the intent in its intent table. We give the code for the callee's actions in Appendix A of our tech report [45].

Beldi handles these cases by first checking B_1 and B_2 with conditional writes before covering the other states exactly as in the unconditional-write case. We give a detailed description in Appendix A of our extended technical report [45].

4.5 Invocation of SSFs and local functions

Finally, Beldi supports three types of function invocations: synchronous calls (`syncInvoke`), which block and return a value; asynchronous calls (`asyncInvoke`), which return immediately; and calls to functions that do not use Beldi's API (e.g., legacy libraries or legacy SSFs). In the first two cases, Beldi guarantees exactly-once semantics. In the last, it only guarantees that the operation is performed at least once.

Figure 8 shows pseudocode for synchronous SSF invocations. As mentioned in Section 3.3, to help SSFs that are being invoked ("callees") differentiate between re-executions and new executions, Beldi passes an instance id to the callee (the "callee id") along with the parameters of the call. In the first invocation, the callee id is generated using `UUID()`; for re-executions, it retrieves the id from the invoke log. If there is already an entry in the invoke log for this caller id and step number, there are two cases: (1) a result is already present, in which case the caller reuses that result; or (2) the entry is present but there is no result, in which case the caller re-invokes the callee with the existing callee id.

Callbacks. Note that `syncInvoke` (Figure 8) does not log the result of the actual call or otherwise mark the call as complete. To see why this is important, consider the example trace in Figure 9, which shows the result of a failure of the callee (SSF2) after it marks itself as done in the intent table

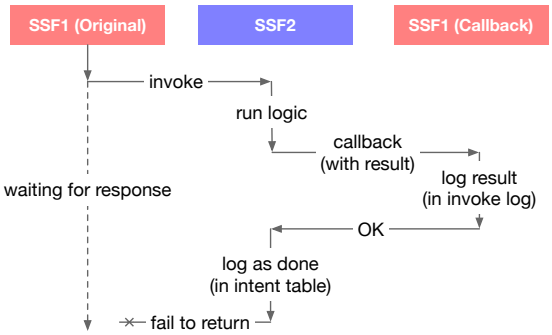


FIGURE 9—SSF1 synchronously invokes SSF2, which then fails to return after logging the operation as done. The callback ensures that SSF1 has the result of SSF2 before SSF2 marks itself as done.

but before it returns the result to the caller (SSF1). Suppose that there is no callback, i.e., that SSF2 logs itself as complete immediately after completing execution. Beldi's federated setup means that each SSF has a garbage collector running at its own pace. If SSF2 were to fail after logging itself as done, it is, therefore, possible that SSF2's GC will garbage collect the intent before SSF1 gets any value. Later, when SSF1's IC re-executes the unfinished SSF1 instance, the caller will see the lack of result in the invoke log, re-invoke SSF2 (with the existing callee id), and SSF2 will mistakenly perform the operation again. In some ways, this is similar to why `write` operations in Beldi must be atomically logged and executed (§3.1). Unfortunately, there are no mechanisms for atomically logging into a database and executing other SSFs.

We address this issue by decomposing an invocation into two steps: (1) the invocation itself, performed by the caller; and (2) the recording of results, done via a second, automatic invocation by the callee to *some* instance of the caller. We emphasize 'some' and 'original' because request routing in serverless is stateless: if SSF1 invokes SSF2, and SSF2 then invokes SSF1, the two SSF1 instances could be different (§2.1). We call this automatic invocation a *callback*. When the second instance of the caller receives the callback, it logs the provided result in its invoke log and returns. At this point, it is safe for the callee to mark its intent as done since it knows the caller's invoke log already contains the result. Note that callbacks only require at-least-once semantics, so there is no need for additional logging of the callback invocation.

Figure 9 illustrates the idea of Beldi's callback mechanism. The callback ensures that the result of SSF2 is properly received by SSF1. As such, we note that SSF2's response to SSF1 is merely an optimization and not necessary for correctness. We also note that if SSF2 fails after a successful callback but before logging the completion of the intent, it may result in a case where SSF1 completes, gets garbage collected, and then a re-execution of SSF2 invokes a spurious callback. SSF1 can detect and ignore this case when a callback occurs for an invoke that does not exist.

Asynchronous invocations. This procedure is similar to that of synchronous invocations, but with the two steps flipped

on the callee. The caller first makes a `rawSyncInvoke` call to the callee, but rather than execute the function, the callee (observing an 'async' flag) simply registers the intent in its intent table, issues a callback, and then immediately returns to the caller. In the second step, Beldi performs the actual asynchronous invocation of SSF2's logic. We describe this operation in detail in Appendix A of our tech report [45].

5 Garbage Collection

If left alone, the linked DAAL will grow indefinitely. While Beldi's use of scans means that the linked DAAL's length is generally not the performance bottleneck, unbounded growth of the linked DAAL and logs (intent table, read log, invoke log) can lead to significant overheads and storage costs. Beldi ensures that logs are pruned and the linked DAAL remains shallow with a garbage collector (GC) that deletes old rows and log entries without blocking SSFs that are concurrently accessing the list. The GC is an SSF triggered by a timer.

At a high level, the protocol has six parts. First, the GC finds intents that have finished since the last time a GC instance ran and assigns them the current time as a finish timestamp. Second, the GC looks up all intents whose finish timestamp is 'old enough' (we expand on this next), and marks them as 'recyclable.' Third, the GC removes log entries (in the read and invoke logs) that belong to recyclable intents. Fourth, the GC disconnects, for every item, the non-tail rows of their linked DAAL that have empty logs, marks these rows as 'dangling', and assigns them the current time as a dangling timestamp. Fifth, the GC removes all rows whose dangling timestamp is 'old enough.' Finally, the GC removes the log entries from the intent table. The algorithm is given in Figure 10, with more details in Appendix A of our tech report [45]. Note that GCs only need at-least-once semantics to avoid memory leaks in the presence of crashes; they do not use Beldi's exactly-once API. Instead, GCs defer the removal of entries in the intent table until the end.

Assumption. The safety of garbage collection relies on a synchrony assumption. In particular, it assumes that an individual SSF instance terminates, one way or another, in at most T time. This allows the GC to delete the logs of completed intents after waiting T time for all running instances of the completed intents to finish. Note that no new instances will be started by an SSF's IC after the intent is marked as 'done.'

Our assumption is based on the observation that serverless providers enforce user-defined execution timeouts on SSF instances (§2.1), but otherwise provide no interface for developers to kill or stop running functions. We can derive a conservative bound for T from these user-defined timeouts. Note that even if providers refuse to kill SSFs after the timeout, we can work around this issue (at high cost) by having the GC change the database's permissions or rename tables so that ongoing SSF instances (including stragglers that stick around after the intent is done) fail to corrupt the database; instances that start after the change are fine.

```
def garbageCollection():
    time = now()
    recyclable = []
    for id, intent in IntentTable:
        if intent[Done]:
            if FinishTime not in intent:
                intent[FinishTime] = time
            elif time - intent[FinishTime] > T:
                recyclable.append(id)
    for id in recyclable:
        remove from ReadLog
            where "LogKey[Id] == {id}"
        remove from InvokeLog
            where "LogKey[Id] == {id}"
    for table, key in getAllDataKeys():
        rows = rawScan(table,
                       cond: "Key == {key}")
        for row in rows:
            for log in row[RecentWrites]:
                mark if log[Id] in recyclable
            if fullyMarked(row[RecentWrites])
                    and row[NextRow] exists:
                prev(row)[NextRow] = row[NextRow]
                if DangleTime not in row:
                   row[DangleTime] = time
        rows = rawScan(table, cond: "Key == {key}
                    && {time} - DangleTime > T")
        for row in rows:
            if row not reachable from head(key)
                delete row
    for id in recyclable:
        remove from IntentTable
            where "LogKey[Id] == {id}"
```

FIGURE 10—Pseudocode for Beldi's lock-free, thread-safe garbage collection algorithm. T is the maximum lifetime of an SSF instance.

Safety of concurrent access. With the above assumption, Beldi's GC preserves exactly-once semantics without needing to interrupt SSF instances. First, observe that an intent is marked as recyclable only after Beldi is sure that no live SSF instance requires the intent. Accordingly, the read log, invoke log, and intent table entries for the intent will never be accessed again. For the linked DAAL, the GC only disconnects a row when all of the contained logs are marked as recyclable and it is not the tail. New traversals of the linked DAAL for read or write operations will not observe the disconnected row (technically the `rawScan` operation will return these disconnected rows, but they will be ignored during the traversal of the local linked DAAL). Running SSF and GC instances, however, may be in the process of traversing the disconnected row—if Beldi deleted it immediately, the SSF or GC might become stranded. To prevent this, Beldi keeps the disconnected row for an additional T time to ensure that instances with such references terminate successfully.

Safety of concurrent modifications. The linked DAAL also supports garbage collection in the presence of concurrent appends from SSFs and deletions from other GC instances ow-

ing to it being a type of non-blocking linked list. In fact, it is simpler than traditional non-blocking linked lists [21, 42, 47] because new rows are always appended to the tail, and GCs never touch the tail. The only interesting case is the concurrent disconnection of neighboring rows such as X and Y in $A \rightarrow X \rightarrow Y \rightarrow B$. In this case, the disconnection of X succeeds, but the disconnection of Y will not be visible because the updated `NextRow` pointer in X is no longer part of the linked DAAL. The next GC run disconnects Y permanently.

6 Supporting Locks and Transactions

In addition to exactly-once semantics, Beldi also provides support for locks and transactions with user-generated aborts.

6.1 Locks

Beldi's approach to mutual exclusion borrows an abstraction in Olive called "locks with intent", where locks over data items are owned by an intent rather than a specific client. This means that, if an SSF instance calls `lock(item)` and then crashes, the lock is not lost and held indefinitely; rather, the IC will soon restart the instance. The re-executed instance, upon arriving at the `lock(item)` call, will see that it already acquired the lock and be able to continue with the remaining operations as if the original SSF instance had never crashed.

In Beldi, the ownership of a lock on a given item is kept alongside the data and logs in the "lock owner" column of the item's linked DAAL. Lock acquisition and release are logged to the DAAL as writes to the item using Beldi's `condWrite` semantics, where the condition is that the lock is either owned by the current SSF or has an empty lock-owner column in the DAAL. The exactly-once semantics are needed for cases where an SSF is re-run after successfully releasing a lock.

Note that Beldi only guarantees exactly-once semantics—it does not absolve the developer from writing bug-free code. Thus, problems like infinite loops within critical sections and deadlock need to be handled with higher-level mechanisms (like the one below) if the user wishes to guarantee liveness.

6.2 Transactions

Beldi uses an extension of the locking mechanism of the preceding section to implement transactions within and across SSF boundaries. Beldi transactions are based on a variant of 2PL with wait-die deadlock prevention and two-phase commit. Note that the choice of wait-die (rather than something like wound-wait) is deliberate as SSF instances generally cannot kill other instances. To implement this, we need to track the intent-creation time of each SSF. We do so by adding to the lock-owner column an intent-creation timestamp and checking upon lock-acquisition failure whether the existing lock owner is older or younger than the current SSF instance; if older, abort, otherwise, try again (see Figure 11).

There are three main parts to Beldi's transaction-handling protocol: (1) creating and forwarding a *transaction context*, (2) executing Beldi calls inside a transaction, and (3) propagating abort/commit signals throughout a workflow. Note that Beldi does not currently support `asyncInvoke` in transactions; however, it does support spawning threads that issue `syncInvoke` operations and are then joined.

```
def lock(table, key):
  ok = condWrite(table, key,
       cond: "LockOwner = NULL
              || LockOwner.id = TXNID",
       update: "LockOwner = [TXNID, START_TIME]")
  if not ok:
    row = read(table, key)
    ownerId, ownerTime = row[LockOwner]
    if ownerTime <= TXNID:
      abort
    else:
      lock(table, key)
```

FIGURE 11—Pseudocode for the lock operation with wait-die deadlock prevention used during the 'Execute' mode of a transaction.

Transaction contexts. In Beldi, transactions are defined with the `begin_tx` and `end_tx` API calls. Beldi assumes that both the begin and the end statements are placed in the same SSF, but SSFs can invoke other SSFs inside a transaction, so transactions can span across multiple SSFs. When an SSF calls `begin_tx` it creates a new top-level *transaction context* which consists of a unique transaction id and a mode ('Execute', 'Commit', or 'Abort'). Contexts start in 'Execute' mode. The SSF instance will also, upon creating a new context, execute the transaction's operations in a new thread/goroutine to catch any runtime exceptions. The matching `end_tx` waits for the result and runs either a commit or abort protocol depending on the outcome of the contained operations.

Transaction contexts are passed along with any SSF invocations that occur inside the transaction. Thus, whenever a Beldi-enabled SSF starts, it first determines whether it is a part of an ongoing top-level transaction by checking whether a context was provided as part of the input. This is necessary even if the SSF never creates a transaction itself. If the SSF does create a transaction, the `begin_tx`/`end_tx` statements will be ignored and all operations will be inherited by the top-level transaction context. Beldi does not currently support *nested transaction* semantics [31] (e.g., a sub-transaction can abort without causing the top-level transaction to abort).

Opacity. Beldi chooses *opacity* as the isolation level for transactions. Opacity [20] captures strict serializability [5, 34] with the additional requirement that even transactions that abort do not observe inconsistent state. The rationale is that observing inconsistent state can lead to undefined behavior and infinite loops. For example, if an SSF instance reads inconsistent state that results in division by zero, it may crash. Beldi's IC will restart the SSF instance and deterministically replay the (inconsistent) values to ensure exactly-once semantics, re-triggering the crash. Figure 12 gives another example of how OCC [26], which provides serializability but not opacity,

```
begin_tx()
x = read("x"); y = read("y")
while (x != y):
  // some logic
  x++
write("x", x + 2); write("y", y+4)
end_tx()
```

FIGURE 12—OCC leads to an infinite loop when two instances of the above transaction, T_1 and T_2, execute concurrently. Suppose $x = 0, y = 1$ initially. T_1 reads $x = 0, y = 1$, executes the logic, acquires locks on x and y, validates the read set, and writes $x = 3, y = 4$. T_2 reads $x = 3, y = 1$ (corresponding to a state after which T_1 updated x but before it updated y), and is stuck in an infinite loop. Even though T_2 is destined to abort, it will never reach the read set validation step.

leads to infinite loops. These issues are not present with isolation levels that guarantee that all transactions read from a consistent snapshot.

Operation semantics inside a transaction. If an SSF is in a transactional context, Beldi modifies the semantics of its API based on the mode to ensure ACID semantics. We have already discussed two operation modifications that occur in 'Execute' mode—one to locks in Figure 11 and another to `begin_tx/end_tx`, which are ignored. 'Execute' mode also causes Beldi to call `lock` before every `read`, `write`, and `condWrite` operation, using the transaction id as the lock holder. In addition to acquiring locks, Beldi also changes where reads and writes look up and record values. While lock acquisition still goes to the original tables, Beldi redirects written values to a *shadow table* that acts as a local copy of state for the transaction. Like the original table, this shadow table is also stored as a linked DAAL and is garbage collected along with the normal DAAL (except the GC also deletes the head and tail). Unlike the original, the shadow table is partitioned by transaction id, with `Key` relegated to a secondary index. All `read` operations check the shadow table first before consulting the real table to ensure that transactions read their own writes. If, before an operation, an SSF fails to acquire a lock and must kill itself (due to wait-die), it returns to its caller with an 'abort' outcome.

Propagation of commit or aborts. Eventually, a `begin_tx/end_tx` code block will reach the `end_tx` with an abort/commit decision. For commit, Beldi changes the mode of the context to 'Commit', flushes the final values of the items in the shadow table to the real linked DAAL, and releases any held locks. Beldi then calls the SSF's callees and passes them the transaction context in Commit mode. Note that if an SSF instance fails between flushing the shadow table and notifying the callees of the commit decision, Beldi's exactly-once semantics ensure that once the SSF instance is re-executed, it will pick up from where it left off. For abort, none of the values have been written to the actual table, so Beldi just releases all locks and invokes all callees in 'Abort' mode.

When an SSF is invoked with a transaction context that includes a Commit mode, Beldi skips the SSF's logic, and instead performs only the aforementioned commit protocol: flushes the final value of the items, releases any held locks, and notifies its own callees by invoking them with the provided transaction context. An Abort mode similarly skips the SSF's logic, releases all locks, and notifies its callees. This recursive invocation of callees with a Commit or Abort mode mimics the role of a coordinator in two-phase commit.

Supporting step functions. The previous discussion assumes a `begin_tx` and `end_tx` in the same SSF. To support transactions across SSFs defined in *step functions*, the developers must introduce 'begin' and 'end' SSFs in their workflow (we give an example in Appendix A of our tech report [45]). These SSFs create the transaction context and kickstart the commit or abort protocol. SSFs that fall between the 'begin' and 'end' SSFs in the workflow execute transactionally. If an SSF aborts it sends 'abort' on its outgoing edges in the workflow; an SSF that receives an abort as input skips its operations and propagates the abort message on its outgoing edges. This continues until the abort message reaches the 'end' SSF, which then sets the transaction context mode to Abort and invokes the 'begin' SSF. If 'end' executes without receiving any abort message, it sets the context mode to Commit instead. This invocation initiates the second phase of 2PC over the transactional subgraph of the workflow.

Non-transactional SSFs inside transactions. While an SSF that does not use transactions can be invoked inside a transaction by another SSF (which automatically forces the non-transactional SSF to acquire locks before any accesses), app developers must ensure that the non-transactional SSF is *only* used inside transactional contexts. Otherwise, non-transactional instances may access the database without acquiring locks or obeying the wait-die protocol.

7 Evaluation

Beldi brings forth an array of programmability and fault-tolerance benefits, but with these benefits come costs. In this section we are interested in answering three questions:

1. What is the cost of maintaining and accessing the linked DAAL, and how does it compare to applicable baselines?
2. What are the latency and throughput of representative applications running on Beldi, and how does Beldi compare to existing serverless platforms that provide neither exactly-once semantics nor transactional support?
3. What effect does Beldi's GC have on linked DAAL traversal, and how does it change as we adjust the timeout (T)?

We answer the above questions in the context of the following implementation, applications, and experimental setup.

7.1 Implementation

We have implemented a prototype of Beldi for Go applications that runs transparently on AWS Lambda and DynamoDB. In

total, Beldi's implementation consists of 1,823 lines of Go for the API library and the intent and garbage collectors.

Case studies. To evaluate Beldi's ability to support interesting applications at low cost, we implement three case studies: a social media site, a travel reservation system, and a media streaming and review service. We adapt and extend these applications from DeathStarBench [12, 16], which is a recent open-source benchmark suite for microservices, and port them to a serverless environment (using Go and AWS Lambda). This port took around 200 person-hours. Combined, our implementations total 4,730 lines of Go. We provide details of the corresponding workflows in Appendix B of our tech report [45], and give a brief description below.

Movie review service (Cf. IMDB or Rotten Tomatoes): Users can create accounts, read reviews, view the plot and cast of movies, and write their own movie reviews and articles. Our implementation of this app consists of a workflow of 13 SSFs.

Travel reservation (Cf. Expedia): Users can create an account, search for hotels and flights, sort them by price/distance/rate, find recommendations, and reserve hotel rooms and flights. The workflow consists of 10 SSFs, and includes a cross-SSF transaction to ensure that when a user reserves a hotel and a flight, the reservation goes through only if both SSFs succeed. Note that we extend this app to support flight reservations, as the original implementation [12] only supports hotels.

Social media site (Cf. Twitter): Users can log in/out, see their timeline, search for other users, and follow/unfollow others. Users can also create posts that tag other users, attach media, and link URLs. The workflow consists of 13 SSFs that perform tasks like constructing the user's timeline, shortening URLs, handling user mentions, and composing posts.

7.2 Experimental setup

We run all of our experiments on AWS Lambda. We configure lambdas to use 1 GB of memory and set DynamoDB to use autoscaling in on-demand mode. All of the read and scan operations for Beldi and the baseline use DynamoDB's strong read consistency. We turn off automatic Lambda restarts and let Beldi's intent collectors take care of restarting failed Lambdas. Our garbage and intent collectors are triggered by a timer every 1 minute, which is the finest resolution supported by AWS. Note that AWS currently has a limit of 1,000 concurrent Lambdas per account. As we will see in some of our experiments, this limit is often the bottleneck in both the baseline and Beldi. Finally, consistent with our deployability requirement (§2.2), Beldi uses no servers.

The baseline for our experiments is running our ported applications on AWS Lambda without Beldi's library and runtime. Consequently, these applications will not enjoy exactly-once semantics or support transactions: when running on the baseline, the travel reservation app outputs inconsistent results, and all apps can corrupt state in the presence of crashes.

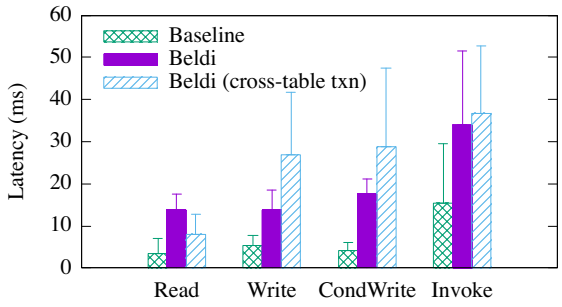


FIGURE 13—Median latency of Beldi's operations. Error bar represents the 99th percentile, and "cross-table tx" is an implementation of Beldi that uses cross-table transactions instead of the linked DAAL.

7.3 What are the costs of Beldi's primitives?

We start our evaluation with a microbenchmark that measures the cost of each of Beldi's primitive operations: `read`, `write`, `condWrite`, and `invoke`. The keys are one byte and the values are 16 bytes. We measure the median and 99th percentile completion time of the four operations over a period of 10 minutes at very low load (1 req/s). As baselines, we also measure the completion time (1) without Beldi's exactly-once guarantees and (2) using a design that logs writes to a separate table using cross-table transactions. Since Beldi's database operations depend on the length of the linked DAAL, we populate the chosen key's linked DAAL with a conservative value of 20 rows, which corresponds to the length of the linked DAAL after 30 minutes without garbage collection as described in the experiment of Section 7.5.

Figure 13 shows the overhead of Beldi's reads/writes compared to those of the baseline stem from two sources: scanning the linked DAAL (instead of reading a single row) and logging. For invoke, the overheads come from our callback mechanism and logging to the invoke log. Consequently, all of Beldi's operations are around 2–4× more expensive than the baseline. In contrast, the approach using cross-table transactions does not use a DAAL so `read`s avoid the scan (but not the logging), and `write`s perform an atomic transaction where the value is written to one table and the log entry is added to another. The cost of this operation is 2–2.5× higher than Beldi's linked DAAL. Appendix C in our tech report [45] describes the same experiment with a more optimistic setting (5 rows in the linked DAAL); the results are similar.

Note that not all existing databases (e.g., Bigtable) support cross-table transactions. Even for those that do, the performance gain that cross-table transactions have on read operations over using a linked DAAL goes away whenever SSFs use transactions because read locks use `condWrite` which is a cheaper operation on the linked DAAL.

Other costs. Another consideration beyond performance is the additional storage and network I/O required by Beldi to maintain and access all logs and linked DAAL metadata. For our setup above, the 20-row DAAL for the item takes up

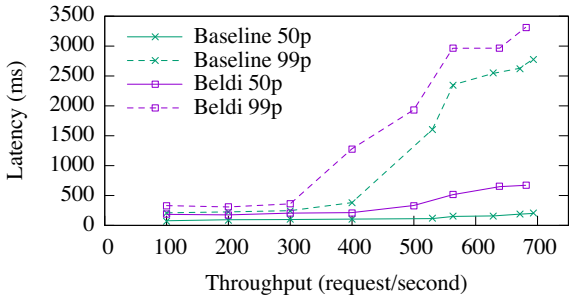


FIGURE 14—Median response time and throughput for our movie review service. Dashed lines represent 99th-percentile response time.

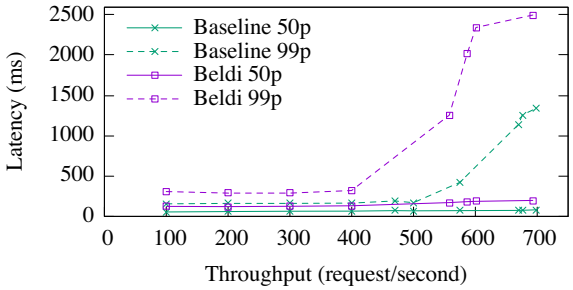


FIGURE 15—Median response time and throughput for travel reservation service. Dashed lines represent 99th-percentile response time. Beldi performs transactions over multiple SSFs to reserve a hotel room and a flight, while the baseline returns inconsistent results.

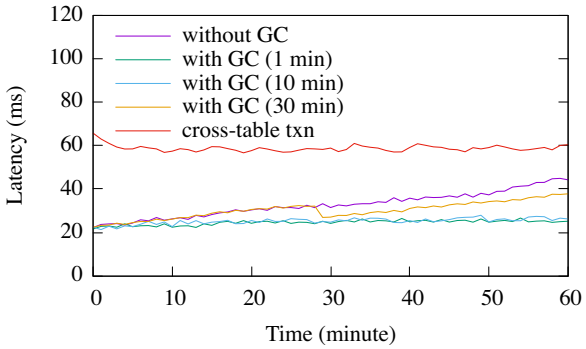


FIGURE 16—Median response time for an SSF that uses one `write` operation under different GC configurations. Without GC, the linked DAAL grows over time. As a baseline, we configure Beldi with cross-table transactions that do not use a linked DAAL.

8 MB of storage. Counting all logs and metadata, each operation requires storing between 20 to 36 bytes in addition to the value. In terms of the network overhead introduced by the scan and projection approach that we use to traverse Beldi's linked DAAL, for a 20-row DAAL, each scan fetches 2 KB more data than a baseline read to a single cell when measured at the network layer. Compared to the baseline, Beldi induces one extra scan and write for each read operation, at least one scan for an unconditional write (and potentially more scans and writes depending on the scenario), and one read and two writes for a function invocation. In DynamoDB's on-demand mode, each read costs an additional $\$2.5 \times 10^{-7}$, whereas writes cost an additional $\$1.25 \times 10^{-6}$. In provisioned-capacity mode, costs are lower but depend on the specified capacity.

7.4 How does Beldi perform on our applications?

In this section, we discuss the results of our large-scale experiments for the movie review and travel reservation services; the social networking site has similar results, so we defer its results to our tech report [45]. The workloads that we use are adapted from DeathStarBench [12, 16] with a minor modification to support our extended travel reservation service: the transactions to reserve a hotel and flight randomly pick a hotel and a flight out of 100 choices each following a normal distribution. Requests contain random content within the expected schema and are generated and measured using wrk2 [44].

We issue load at a constant rate for 5 minutes, starting at 100 req/s and increasing in increments of 100 req/s until the system is saturated. For our applications, we achieve saturation at around 800 req/s. The primary bottleneck in all cases is compute: AWS enforces a limit of 1,000 concurrent Lambdas per account (even if the Lambdas are for different functions), and the HTTP Gateway (or some internal scheduler) rejects requests in excess of this limit.

Figures 14 and 15 depict the results. In all cases (including the social media app), we observe that, until around 400 req/s (34M per day), Beldi's median and 99th-percentile response time are each around 2× higher than that of the baseline. At the highest loads that we could test on AWS, Beldi still achieves the same throughput as the baseline at a slightly higher median response time (around 3.3× for the travel reservation). At this high load, Beldi's 99th-percentile latency is only 20% higher for the movie review service, and 80% higher for the transaction-enabled travel site. We also test a configuration of the travel site that uses Beldi for fault-tolerance but without transactions. The median latency at saturation for that configuration is 16% lower and the 99th-percentile latency is 20% lower than Beldi with transactions.

7.5 What is the effect of garbage collection?

Finally, we evaluate the importance of the choice of garbage collector timeout (T) on performance. Note that this is different from the 1-minute timer that triggers the GC SSF (§7.2). T is instead proportional to the maximum lifetime of an SSF and determines when a GC can remove a row from the Linked DAAL. Thus, this value is important for safety, whereas the trigger only determines when the GC runs.

Since T is important to ensure exactly-once semantics, we could imagine performing a similar actuarial analysis to those involved in setting the end-to-end timeouts of reliable failure detectors [1, 27]. However, as Figure 16 shows, the median response times for SSFs that access the linked DAAL are only lightly impacted by the choice of T, even as we run the system for 30 minutes at constant load under pessimistic conditions (all SSF instances write to the same key). As a result, we

can be relatively conservative about T. To be clear, this is a testament to the heroic efforts of DynamoDB engineers that have optimized its scan, filter, and projection operations. Nevertheless, we take some slight credit for ensuring that Beldi's linked DAAL is compatible with such operators.

It is worth noting, however, that while T has a minor impact on performance, it does impact storage overhead and I/O, since `read` and `write` operations still fetch a projection of the linked DAAL which scales with the number of rows (§7.3).

8 Discussion

We now discuss a few aspects of Beldi, such as the implication of relying on strongly consistent databases, the potential benefit of using SQL databases like Amazon Aurora, and the security implications of SSF federation and reusability.

Strongly consistent databases. Beldi enables developers to write stateful serverless applications without having to worry about concurrency control, fault tolerance, or manually making all of their functions idempotent. In doing so, Beldi leverages one or more fault-tolerant databases configured to be strongly consistent. If these databases were to become unavailable, for example due to network partitions, SSFs that write to these unavailable databases would also become unavailable until the partition was resolved.

ACID databases. A natural question is whether SSFs that use ACID databases need all of Beldi. For such SSFs, the benefit is not having to maintain a read or write log (or a linked DAAL) since the database does its own logging. However, ACID databases are not enough to guarantee exactly-once semantics for function invocations since they provide atomicity for read and write operations, but have no support for invocations. As a result, Beldi would still need to implement mechanisms such as callbacks (§4.5) to ensure that a failed SSF is not mistakenly re-executed despite independent garbage collectors. Furthermore, workflows that contain transactions *across* SSFs would still need a collaborative coordination protocol such as the one proposed in Section 6.2.

Independence of separate applications. We view SSFs as owning all the data on which they operate, similar to microservice architectures [11]. SSFs can isolate the state of different applications by storing each application's state on a different database. To ensure that a malicious request from one application cannot observe the state of another, standard authentication mechanisms such as capabilities and public key encryption could be used.

9 Related Work

We already discuss Beldi's differences with Olive [36] throughout. To summarize, Beldi builds upon Olive's elegant approach to fault tolerance and mutual exclusion, and adapts it to an entirely new domain. This adaptation is nontrivial and requires us to introduce new data structures, algorithms, and abstractions (e.g., transactions across SSFs). The result of our innovations is a simple API that SSF developers can use to build exciting applications without worrying about fault tolerance, concurrency control, or managing any infrastructure!

In the context of serverless, the observation that existing designs are currently a poor fit for applications that require state has been the subject of much prior work [15, 22, 24, 25, 43]. For example, Cloudburst [40] proposes a new architecture for incorporating state into serverless functions, and `gg` [15] proposes workarounds to state-management issues that arise in desktop workloads that are outsourced to thousands of serverless functions. However, the general approach to fault-tolerance in these works is to re-execute the entire workflow when there is a crash or timeout—violating exactly-once semantics if any SSF in the workflow is not idempotent.

AFT [39] is the closest proposal to Beldi and introduces a fault-tolerant shim layer for SSFs. However, AFT's deployment setting, guarantees, and mechanisms are very different. First, Beldi runs entirely on serverless functions, whereas AFT requires servers to interpose and coordinate all database accesses. As a result, Beldi can run on any existing serverless platform (or even in a multi-provider setup) without requiring any modification on their part and without the user needing to administer their own VMs. Second, Beldi seamlessly enables transactions within SSFs and across workflows with opacity, whereas AFT targets the much weaker (but more efficient) read atomic isolation level [4]. Due to the weaker isolation, it would be more difficult to implement our travel reservation system on AFT. Finally, Beldi allows SSFs to be managed independently and to keep their data private from each other, while AFT's servers manage all SSF data, handle failures and garbage collection, and serve as a central point of coordination for transactions.

10 Conclusion

Beldi makes it possible for developers to build transactional and fault-tolerant workflows of SSFs on existing serverless platforms. To do so, Beldi introduces novel refinements to an existing log-based approach to fault tolerance, including a new data structure and algorithms that operate on this data structure (§4.1), support for invocations of other SSFs with a novel callback mechanism (§4.5), and a collaborative distributed transaction protocol (§6). With these refinements, Beldi extracts the fault tolerance already available in today's NoSQL databases, and extends it to workflows of SSFs at low cost with minimal effort from application developers.

Acknowledgments

We thank the OSDI reviewers for their feedback and our shepherd, Jay Lorch, for going above and beyond and providing suggestions that dramatically improved the content and presentation of our work. We also thank Srinath Setty for many invaluable discussions and his help with Olive. This work was funded in part by VMWare, NSF grants CNS-1845749 and CCF-1910565, and DARPA contract HR0011-17-C0047.

References

[1] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[2] AWS Lambda. `https://aws.amazon.com/lambda/`.

[3] Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`.

[4] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the ACM SIGMOD Conference*, June 2014.

[5] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[6] Cloud Bigtable overview of replication. `https://cloud.google.com/bigtable/docs/replication-overview`.

[7] Quotas and limits for Cloud BigTable. `https://cloud.google.com/bigtable/quotas`.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[9] Consistency levels in Azure Cosmos DB. `https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels`.

[10] Azure Cosmos DB service quotas. `https://docs.microsoft.com/en-us/azure/cosmos-db/concepts-limits`.

[11] C. de la Torre, B. Wagner, and M. Rousos. *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Developer Division, .NET and Visual Studio product teams, v3.1 edition, Jan. 2020. `https://docs.microsoft.com/en-us/dotnet/architecture/microservices/`.

[12] DeathStarBench. `https://github.com/delimitrou/DeathStarBench/`.

[13] Amazon DynamoDB read consistency. `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html`.

[14] Limits in DynamoDB. `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html`.

[15] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[16] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2019.

[17] Google Cloud Functions. Retrying background functions. `https://cloud.google.com/functions/docs/bestpractices/retries`.

[18] Google Cloud Functions. `https://cloud.google.com/functions`.

[19] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, 1978.

[20] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2008.

[21] T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, Oct. 2001.

[22] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research (CIDR)*, Jan. 2019.

[23] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[24] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. Formal foundations of serverless computing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Oct. 2019.

[25] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[26] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2), June 1981.

[27] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the FALCON spy network. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[28] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Aug. 2013.

[29] .Net Microservices Sample Reference Application. `https://github.com/dotnet-architecture/eShopOnContainers`.

[30] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1), 1992.

[31] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Massachusetts Institute of Technology, 1981.

[32] S. Mu, S. Angel, and D. Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2019.

[33] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating

concurrency control and consensus for commits under conflicts. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[34] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), Oct. 1979.

[35] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[36] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[37] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro. Fast general distributed transactions with opacity. In *Proceedings of the ACM SIGMOD Conference*, 2019.

[38] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, Sept. 2006.

[39] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2020.

[40] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. arXiv:2001/04592, Jan. 2020. `https://arxiv.org/abs/2001.04592`.

[41] AWS Step Functions. `https://aws.amazon.com/step-functions/`.

[42] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, Aug. 1995.

[43] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[44] wrk2: A constant throughput, correct latency recording variant of wrk. `https://github.com/giltene/wrk2`.

[45] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows (extended version). arXiv:2010/06706, 2020. `https://arxiv.org/abs/2010.06706`.

[46] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[47] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear. Practical non-blocking unordered lists. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, Oct. 2013.

A Artifact Appendix

A.1 Abstract

Our artifact runs on Amazon AWS Lambda without additional requirements or dependencies. Deploying the code, performing the measurements, generating the plots, and running the benchmarks depend on some third-party frameworks including serverless, gnuplot and wrk2.

A.2 Artifact check-list

- **Program:** Golang
- **Run-time environment:** AWS Lambda
- **Metrics:** Throughput and latency
- **Experiments:** Our serverless port of DeathStarBench
- **Expected experiment run time:** Around 20 hours
- **Public link:** `https://github.com/eniac/Beldi`
- **Code licenses:** MIT License

A.3 Description

A.3.1 How to access

`https://github.com/eniac/Beldi`

A.4 Installation

A.4.1 Set up docker container

1. login to a registry

   ```
   $ docker login
   ```

2. pull the docker image

   for github packages users:

   ```
   $ docker run -it \
   > docker.pkg.github.com/eniac/beldi/beldi:latest
   /bin/bash
   ```

   for docker hub users:

   ```
   $ docker run -it tauta/beldi:latest /bin/bash
   ```

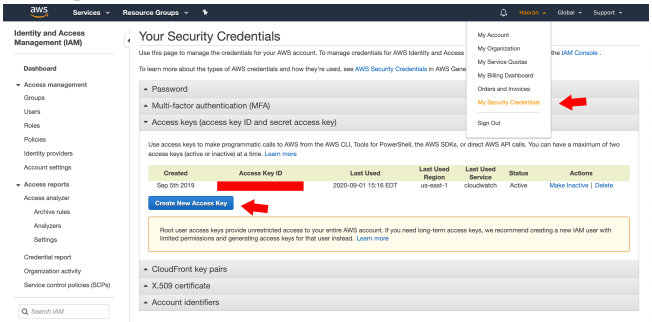
The purpose of this container is to setup the environment needed to run our configuration, deployment, and graph plotting scripts. The actual code of Beldi runs on AWS lambda.

A.4.2 Set AWS Credentials

Inside the container run

```
$ aws configure
```

It will ask you for an access key ID, a secret access key, region and output format. The first two can be found/created at:



Set the region to us-east-1 and the output format to json.

A.5 Evaluation and expected result

A.5.1 Primitives (Figure 13)

To run the experiment

```
$ ./scripts/singleop/run.sh
```

The script has two modes

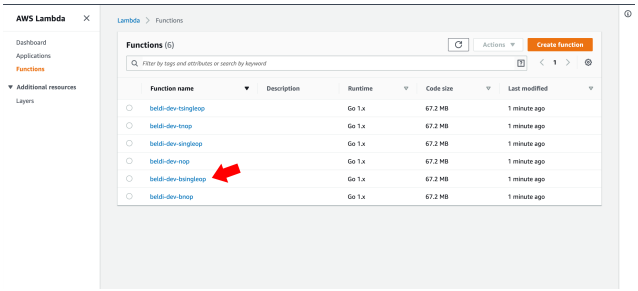
1. fast mode: less time, approximate result (around 5 min)
2. full mode: full experiment (around 30 min)

The script will ask you which mode to run when it starts.

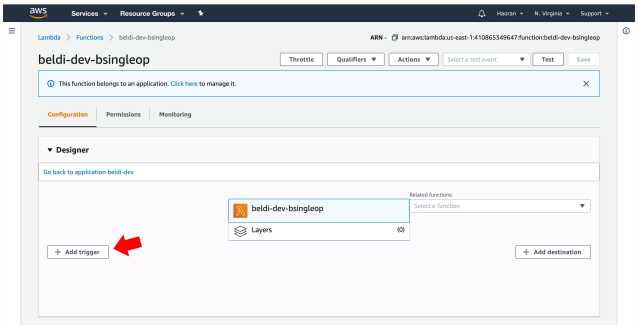
Figure 13 includes three experiments, baseline (without beldi), beldi and beldi-txn. Their function names are bsingleop, singleop and tsingleop respectively. After deployment, the script will ask for the HTTP endpoint for these three lambdas, which needs manual setup at AWS.

Take bsingleop as an example:

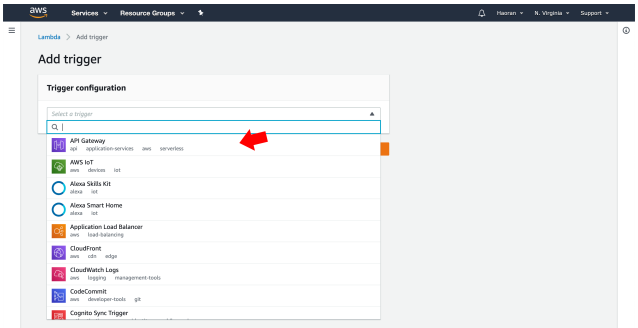
1. Go to the lambda console, click the function



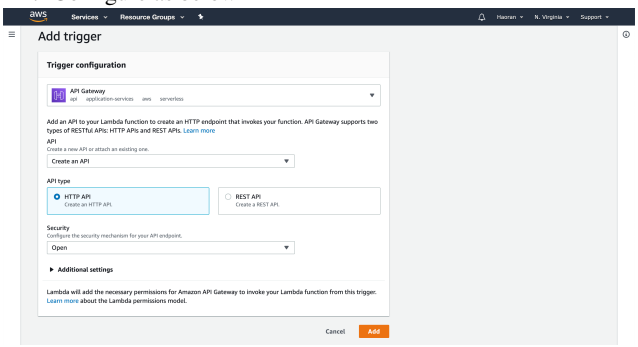
2. Click add trigger



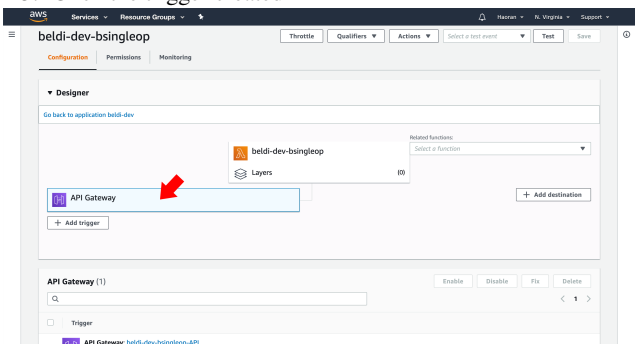
3. Choose API Gateway



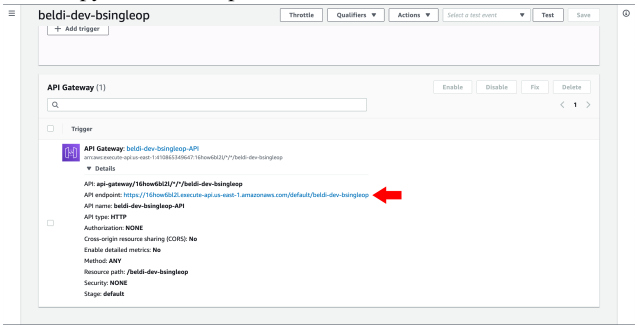
4. Configure as below



5. Click the trigger created



6. Copy the link and paste in terminal



After all three endpoints get set, the experiment will start running. The result will be saved at `beldi/result/singleop/singleop`, which can be loaded by gnuplot

```
$ gnuplot < scripts/singleop/singleop.pg
```

The figure will show up as `beldi/result/singleop/res.png`. You can use docker cp to copy it to your host.

A.5.2 Garbage Collection (Figure 10)

To run the experiment,

```
$ ./scripts/gctest/run.sh
```

The script has two modes

1. fast mode: less time, prefix of Figure 10 (around 30 min)
2. full mode: full experiment (around 150 min)

The script will ask you which mode to run when it starts.

The script compiles the code and deploys the binary to AWS. After that, it will ask for the HTTP endpoint for **beldi-dev-gctest**. The result will be saved as `beldi/result/gctest/gc`.

To generate the figure,

```
$ gnuplot < scripts/gctest/gc.pg
```

The figure will show up as `beldi/result/gctest/res.png`.

A.5.3 Movie review service (Figure 14)

Baseline.

```
$ ./scripts/media/run-baseline.sh
```

Each data point takes around 20 min.

The script will first ask you for a request rate (the default is 100). After deployment, it will ask for the HTTP endpoint for **beldi-dev-bFrontend**. When it finishes, it will print to the terminal the median and p99 latency. This result will also be saved to `result/media/baseline.json`. Alternatively, you can view the metrics on AWS CloudWatch.

Beldi.

```
$ ./scripts/media/run.sh
```

A.5.4 Travel Reservation (Figure 15)

Baseline.

```
$ ./scripts/hotel/run-baseline.sh
```

Each data point takes around 20 min.

It will ask for the HTTP endpoint for **beldi-dev-bgateway**. When it finishes, it will print to terminal the median and p99 latency. It will also save the result to `result/hotel/baseline.json`.

Beldi.

```
$ ./scripts/hotel/run.sh
```

Unearthing inter-job dependencies for better cluster scheduling

Andrew Chung* Subru Krishnan† Konstantinos Karanasos† Carlo Curino† Gregory R. Ganger*
*Carnegie Mellon University †Microsoft

Abstract

Inter-job dependencies pervade shared data analytics infrastructures (so-called "data lakes"), as jobs read output files written by previous jobs, yet are often invisible to current cluster schedulers. Jobs are submitted one-by-one, without indicating dependencies, and the scheduler considers them independently based on priority, fairness, etc. This paper analyzes hidden inter-job dependencies in a 50k+ node analytics cluster at Microsoft, based on job and data provenance logs, finding that nearly 80% of all jobs depend on at least one other job. Yet, even in a business-critical setting, we see jobs that fail because they depend on not-yet-completed jobs, jobs that depend on jobs of lower priority, and other difficulties with hidden inter-job dependencies.

The Wing dependency profiler analyzes job and data provenance logs to find hidden inter-job dependencies, characterizes them, and provides improved guidance to a cluster scheduler. Specifically, for the 68% of jobs (in the analyzed data lake) that exhibit their dependencies in a recurring fashion, Wing predicts the impact of a pending job on subsequent jobs and user downloads, and uses that information to refine valuation of that job by the scheduler. In simulations driven by real job logs, we find that a traditional YARN scheduler that uses Wing-provided valuations in place of user-specified priorities extracts more value (in terms of successful dependent jobs and user downloads) from a heavily-loaded cluster. By relying completely on Wing for guidance, YARN can achieve nearly 100% of value at constrained cluster capacities, almost 2× that achieved by using the user-provided job priorities.

1 Introduction

Data lakes have become core elements of modern data-driven enterprises, providing required data storage and analysis infrastructure (see Fig. 1). Data lakes enhance data processing via a combination of two critical properties: (i) a highly consolidated, multi-tenant infrastructure that enables multiple teams of data scientists and engineers to share resources rather than each having their own, and (ii) low data access barriers that allow easy data sharing between users and various types of data analytics applications. Combined, these properties increase data re-use [4, 27] and reduce overall computational resource-hours consumed [31, 33].

This same data and resource sharing creates a new challenge: hidden inter-job dependencies. We say that Job 2 depends on Job 1 if Job 2 takes as input any output file generated and stored into the shared distributed file system by Job 1.[1] For example, in Fig. 1, Job 3 (from Org 3) depends on Job 2, which in turn depends on Job 1. We refer to these as *hidden* dependencies, to contrast them with explicit computation DAGs managed by schedulers within workflow managers [30,40,41], because there is no indication of such dependencies indicated in the job submissions—the dependencies are not expressed to the cluster scheduler.

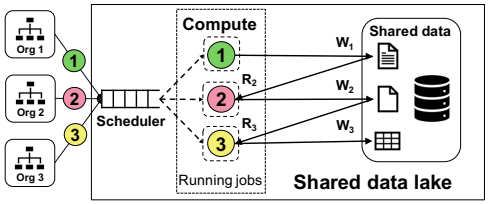


Figure 1: Data lake overview. Different jobs submitted by different organizations share the same compute infrastructure and read (R) and write (W) to the same storage system, thereby creating *inter-job dependencies* as jobs consume the output of other jobs. e.g., Job 2 (from Org 2) reads a file written by Job 1, so Job 2 depends on Job 1.

The advent of GDPR [56] forced large companies such as Microsoft to invest in infrastructures to track data provenance and data movement both within the data lake and to external components. This created an unprecedented opportunity to uncover and exploit these inter-job dependencies for scheduling: We analyze data extracted from petabytes of job and data provenance logs for 90 days of a 50k+ server cluster (part of Microsoft's Cosmos data lake [6, 12]) shared by over 1300 users from more than 150 internal organizations. In total, our analysis covers over 4 million submitted jobs and 16 million inter-job dependencies. We find that almost 80% of submitted jobs depend on output generated by at least one other job. Indicating the breadth of sharing, many dependencies are cross-organization, with 20% of jobs depending on jobs submitted by another organization.

Despite so much inter-job dependence, systems provide little support for addressing associated challenges. For example, in Cosmos, different users and organizations make their own decisions regarding when to submit jobs and how to set job priorities. Ideally, all co-dependent organizations and users would set up clear Service Level Agreements among themselves to ensure timely arrival of input data for business-critical analyses. Yet, we see signs of insufficient coordination

[1] Our nomenclature and analyses focus on fundamental dataflow dependencies among batch analytics jobs, not distributed stream processing or artificial inter-relationships caused by resource contention.

to ensure that jobs' outputs are produced in time for consumption by dependent jobs. For example, 13% of submitted jobs depend on output files from jobs that execute at a lower priority, which can result in priority inversion since job schedulers are not dependency-aware. More broadly, 34% of recurring jobs are submitted without checking if inputs they depend on are available, failing immediately if they are not.

The Wing dependency profiler efficiently processes prior job and provenance data to predict the impact of each new job on future jobs and user downloads. Although it is inherently difficult to know what future jobs will depend on the output generated by a current job, Wing finds success by focusing on recurrence. Previous workload studies have shown that > 60% of jobs in data analytics environments are *recurrent* and suggest that dependencies of these jobs can similarly follow certain patterns [34, 51]. Our analyses in Cosmos confirm that inter-job dependencies are recurrent (79% of all inter-job dependencies are recurrent), with jobs of the same template exhibiting recurring input consumption patterns. As such, Wing uses *historically recurring dependencies* to (i) analyze and predict relationships between common, dependent recurring jobs, and (ii) guide a cluster scheduler to value jobs in a way that accounts for hidden dependencies.

To explore Wing's efficacy, we pair Wing with stock YARN scheduling (*Wing-Agg*), replacing user-provided priorities with Wing-guided priorities. Specifically, we use number of downloads attained associated with a job's outputs as an approximation for job value,[2] and assign priorities to jobs based on *value efficiency* [8, 28, 44] (job-value divided by resource-time-used). We use trace-driven simulation to evaluate Wing-Agg, compared to using the user-provided priorities (as used in Cosmos), when the goal is to maximize the overall value attained. We find that Wing-guided scheduling achieves up to 66% more value than the Cosmos default, under cluster capacity crunch. Further, when organizational cluster resource boundaries are removed, a Wing-guided scheduler can achieve nearly 100% of value at constrained cluster capacities, almost 2× the value achieved by scheduling based on user-provided job priorities.

Contributions. This paper makes three primary contributions: (i) It presents the first detailed public study of hidden inter-job dependencies in a large-scale data analytics cluster, revealing important problems and opportunities; (ii) it describes a novel system for extracting historical inter-job dependencies from provenance data, at scale, and predicting the impact of a newly-submitted job on future jobs and users; (iii) it shows that use of such predictions can allow a modern scheduler, with minimal changes, to better serve the overall workload by prioritizing the highest-impact jobs.

[2] While job output download-counts are imperfect as ground-truth for job value, a limited check (§4.3) against known important levels for six business-critical jobs indicates that it at least sometimes behaves reasonably.

2 Hidden inter-job dependencies in Cosmos

This section describes and analyzes hidden inter-job dependencies in a large production data lake (Cosmos), highlighting observations that affect resource scheduling decisions and opportunities. It provides an overview of Cosmos and inter-job dependencies, introduces terminology used through the rest of the paper, and quantifies the prevalence and characteristics of hidden inter-job dependencies.

2.1 Cosmos

Overview. Cosmos is one of the largest big data analytics infrastructures in the world. Deployed internally within Microsoft, it is made up of multiple clusters, each with 50k+ nodes [12]. Within Cosmos, more than 80% of infrastructure capacity is dedicated to SCOPE jobs [6, 12], which are batch data analytics jobs similar in nature to Apache Spark [57] and MapReduce [14]. Our work primarily focuses on SCOPE jobs and inter-job dependencies between them.

CosmosFS and operations. SCOPE jobs submitted to a Cosmos cluster read input from and write output to a distributed file system known as the *CosmosFS*. A user can also access CosmosFS through a front-end service to upload or download files directly. We call actions performed on files in CosmosFS, either by SCOPE jobs or through the front-end, *operations*.

Continuous logging. Cosmos continuously tracks and logs data provenance and job telemetry (e.g., compute-hours, submission/completion time, and job structure metadata) into external services: *ProvRepo* stores data provenance and and *JobRepo* stores job telemetry. Our analyses and Wing use these logs to figure out inter-job dependencies.

Job template vs job. A job template [32, 34] is a program to be executed (one or multiple times) in Cosmos, while a job is an actual execution of a job template. Each submission of a job template results in a job.

SCOPE job submission patterns. Common patterns used to submit SCOPE jobs within Microsoft include:

(i) *Manual submissions*: Where a job is manually submitted.

(ii) *Workflow managers*: Workflow managers allow users to automate SCOPE job submissions using *workflows*. Workflows consist of inter-dependent jobs that often map to a business task, and can be triggered periodically or conditionally. Within Microsoft, there are at least five major production workflow managers, each with thousands of users.

(iii) *Custom shell scripts*: Scripts can be set up to perform automated job submissions for users. This method is more flexible, but requires specialized management.

2.2 Inter-job dependencies

How are inter-job dependencies formed? We say that a job *A* depends on a job *B* if *A* consumes *any* of *B*'s output as input. As a concrete example of a recurring cross-organization inter-job dependency, periodic jobs deployed by the data compliance team process CosmosFS access logs, which are generated hourly by the CosmosFS team, to detect data compliance

Characteristic	Description	Heuristic
Recurring	*Recurring jobs* are jobs whose template is submitted many times over time, often to analyze fresh data. *Recurring dependencies* are dependencies occurring between jobs of two recurringly-submitted job templates.	Borrowing from Morpheus [34], jobs are identified as *recurring* if (a) jobs of a template are submitted at least three times over a period of three months, with at least one submission each month, (b) templatized job names are an exact match, and (c) source-code signatures are an approximate match. Dependencies are identified as recurring if both the upstream and the downstream jobs are recurring.
Ad-hoc	*Ad-hoc jobs/dependencies* are those not recurring.	*Ad-hoc jobs/dependencies* are those not identified as recurring.
Periodic jobs	*Periodic jobs* are recurring jobs that are submitted "on-the-clock" at a fixed cadence (e.g., submitted every hour at the start of the hour).	Jobs of a template are identified as *periodic* if they are recurring and if job submissions have near-constant inter-arrival time. To determine if inter-arrival times are near-constant, we use the coefficient of variation (CV). Jobs with small CV in their inter-arrival times are identified as *periodic*, while others are *aperiodic*.
Polling	Jobs are *polling* if they scan and wait for their inputs to become available before their submission. Input dependencies of polling jobs are similarly polling.	Jobs are identified as *polling* if they (a) are not identified as periodic, indicating that they are not submitted on a clock, (b) never fail due to missing files from their recurring upstream jobs, and if (c) they are submitted within 15 minutes of the completion of their latest-completing dependent job. Input dependencies of a polling job are *polling*.
Hard dependencies	Dependencies are *hard* if the downstream job requires the output(s) of the upstream job to be able to run successfully. If the input(s) of the downstream job is not ready by the time of its submission, the downstream job fails with a missing file exception.	Dependencies are identified as *hard* if they are (a) ad-hoc, (b) recurring *and* > 95% of jobs of the same template consume the output of only one job of the same upstream job template, or (c) if the downstream job consumes the output of the same number of upstream jobs of the same job template all the time, indicating that they expect the same number of inputs from the same number of jobs from the upstream template.

Table 1: Summary of and heuristics to identify and characterize job and dependency types.

issues. There are many ways inter-job dependencies can form, and while some inter-job dependencies form through careful negotiation between users/organizations, most are formed *organically*, such as via:

(i) *Data discovery through data catalogs*: A user finds an interesting dataset while browsing through Microsoft's internal data catalog, and sets up a job to analyze the dataset.
(ii) *Script inheritance*: A user wanting to submit a SCOPE job to analyze a popular dataset often starts with a script written and shared by others, that contains logic to extract the dataset. The new script, while containing custom logic, often retains parts of the original script (e.g., priority settings).
(iii) *Logically related intra-workflow jobs*: Workflows, which can consist of multiple inter-connected jobs, are often constructed to improve job modularity and manageability. Each run of a workflow potentially creates many inter-job dependencies, as jobs within a workflow are inter-dependent. Note that, although a workflow manager may know about these inter-job dependencies, there is no interface for a workflow manager to express them to Cosmos.

Characteristics of jobs and dependencies. Our analyses uncovered a few major types of dependency and job characteristics based on job submission patterns (Table 1). The three most important job and inter-job dependency characteristics for our purposes are *recurring*, *ad-hoc*, and *hard*.

Challenges. Among the many ways in which inter-job dependencies can form and evolve, most promote loosely maintained (or non-existent) contracts between inter-dependent jobs in favor of developer convenience. This leads to an environment in which most users know little about upstream jobs that produce their input datasets, and even less about downstream jobs that depend on the data their jobs produce. These sub-optimal inter-job dependency configurations are often only exposed as a result of capacity impairment, unexpected job failures, or data/job audits. Indeed, inter-job dependencies are *hidden* through the availability of the many disaggregated solutions to manage and submit jobs and workflows, prompting us to develop Wing to uncover these dependencies.

2.3 Observations on inter-job dependencies

This section motivates our work on exploiting inter-job dependencies by describing consequential empirical observations about our inter-job dependency data, observed over three months in a single Cosmos cluster.

Observation 1 (Recurring jobs & dependencies): Most jobs and dependencies are recurring. Recurring jobs make up 68% of all submitted jobs (the other 32% of jobs are ad-hoc), while recurring dependencies make up 79% of all dependencies (the other 21% of dependencies are ad-hoc). Recurringness of jobs and dependencies suggest predictability, which we show to be achievable in §3.

Observation 2 (Priority mis-configurations): In Cosmos, jobs are assigned resources in declining priority order, where the priority of a job is assigned by the job's submitter. Here, we find that potential priority mis-configurations are frequent within Cosmos: jobs of 21% of job templates have the chance to be systematically priority-inverted—i.e., recurring jobs consuming their output have a higher priority. In addition, up to 33% of ad-hoc jobs are assigned higher priority than the average recurring job submitted within the same hierarchical queue,[3] where recurring jobs are often production jobs [34].

Observation 3 (Uncoordinated jobs): Many jobs are submitted without explicit coordination with respect to the completion of their upstream jobs—i.e., these jobs do not wait for their input to become available nor are tolerant to missing input, yet they are submitted blind with respect to the avail-

[3] *Hierarchical queues* designate resource shares of an organization in clusters at Microsoft. Priorities are only comparable between jobs in the same queue.

ability of their inputs. Such jobs make up 34% of recurring jobs, and can be susceptible to failure due to missing input from an upstream job not completing in time.

Observation 4 (Cross-org jobs & dependencies): Cross-org jobs and dependencies are common at Microsoft. Up to 95% of organizations have cross-org dependencies. Of all dependencies, 33% are cross-org, and 17% of template dependencies are cross-org, where a *template dependency* is a dependency between recurring jobs of two job templates. Furthermore, 28% of jobs and 23% of recurring jobs are involved in cross-org dependencies. Cross-org dependencies can be harder to manage because they require coordination between jobs across hierarchical queues and between job owners across different organizations.

Observation 5 (Jobs are highly inter-connected): Modeling jobs and their dependencies as a directed acyclic graph (DAG), where inter-job dependencies represent edges, we find that more than 50% of jobs are inter-connected in a single weakly connected component (CC), and CCs of sizes ≥ 10 cover more than 80% of all jobs. We also find that the larger a CC, the more bottom-heavy it is—the failure of certain jobs in such large CCs can cause significant amounts of cascading failure downstream.

Observation 6 (Many jobs can be load-shifted in time): Analyzing when the outputs of jobs are consumed by both downstream jobs and users, we find significant opportunity to delay, or load-shift jobs in time, which allows cluster operators to mitigate capacity crunches or reduce power cost [2, 3, 36]. A job has the potential to be able to be load-shifted by up to T hours if it is (1) recurring, (2) has a gap of $> T$ hours before its output(s) are consumed, *and* (3) has run times of $< T$ hours. (1) allows the job to be identifiable in the future, and (2) and (3) ensure that the job has enough slack to be safely delayed by up to T hours. We find that 31, 27, 22, and 14% of all jobs can be potentially load-shifted by up to T = 1, 3, 6, and 12 hours, respectively. [4]

Discussion. We have seen failure due to lacking input and priority inversions happen during manual inspection of job logs and dependency graphs, but we can not provide counts. We have also seen that: (1) users can and do fix their jobs, sometimes at the cost of sub-optimal performance and results, to work around issues, such as by by consuming stale data; and (2) some of these problematic inter-job dependencies can be masked with sufficiently available resources. A better understanding of inter-job dependencies can help us uncover problematic mis-configurations before they show up.

3 Inter-job dependency predictability

Inter-job dependencies show potential in guiding scheduling; but it is unrealistic to expect job submitters to provide all inter-job dependencies up-front due to the fragmented nature of inter-dependency knowledge (§2.2). While *inter-job dependency recurrence* shows promise, for Wing to effectively guide schedulers with inter-job dependencies, recurring inter-job dependencies also need to be *predictable*—i.e., it is important that past dependencies tell us something about the future. In this section, we use a simple model to predict future occurrences of recurring inter-job dependencies, and show that inter-job dependencies can be predictable.

3.1 Prediction model

Given a specific point in time where a job j_u of template J_u ($j_u \in J_u$, where the symbol "∈" is used as shorthand for "of instance") has arrived, for each recurring job template J_d that depends on the output of template J_u in a recurring fashion, our prediction model has two targets: **(T1)** *whether or not* a recurring job $\in J_d$ will arrive and depend on j_u in the future and **(T2)** *when* the first instance of such a job will arrive.

Model for (T1): Will a downstream recurring job arrive? For (T1), our model uses a configurable *prediction threshold* $tr\%$ ranging from 0 to 100 to predict whether or not a job $\in J_d$ will arrive: If $\geq tr\%$ of prior jobs $\in J_u$ have their outputs consumed by a job $\in J_d$, then the predictor predicts `true`; otherwise it predicts `false`.

Model for (T2): When will a downstream job arrive? For (T2), our model aggregates previously observed recurring dependencies where the upstream job $\in J_u$ and the downstream job $\in J_d$, and computes the median elapsed time from the *submission* of the upstream job to the submission of the first dependent downstream job.

3.2 Predictability evaluation

Dependencies change slowly over time. Dependency patterns of recurring jobs change slowly over time, and making predictions based on inter-job dependencies over longer periods of time presents challenges. For example, in (T1), using a month of inter-job dependency data to train our model to predict the arrival of dependent jobs occurring in the next month only allows us to capture at most 77% of upcoming jobs. Regularly training our model on a week of data to predict for the next week works comparatively well, because (1) it allows us to capture up to 95% of upcoming jobs and (2) it allows us to characterize the dependencies of 89% of job templates (covering 97% of all jobs), since jobs of most templates are submitted with an inter-arrival time of less than a week (with daily submissions being the most common).

(T1) metrics and model performance. We evaluate the prediction quality of our model on (T1) based on precision[5] and recall.[6] Fig. 2 examines the tradeoff between precision and

[4] While load-shifting is an interesting topic for future research, we do not directly address methods for load-shifting in this paper.

[5] *Precision* is defined as the number of true positives (TPs) divided by the sum of TPs and false positives. Precision can be thought of as the percentage of positive predictions our model makes (i.e., a downstream job will arrive) that are truly relevant (i.e., such a downstream job actually arrives).

[6] *Recall* is defined as TPs divided by the sum of TPs and false negatives. Recall can be thought of as the percentage of relevant results that our model is able to correctly predict.

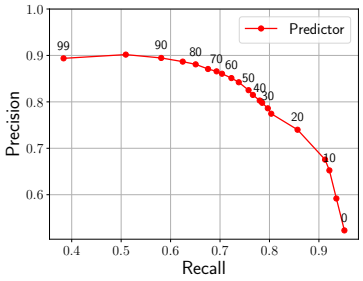


Figure 2: (T1) precision-recall tradeoff. *Predictor* shows the precision-recall tradeoff our dependency-based job arrival predictor makes. Each point on the curve specifies a different setting for the prediction threshold (tr). As $tr \rightarrow 100\%$ (more selective), a larger fraction of predictions are relevant (more precision), but less relevant jobs are captured in total (less recall).

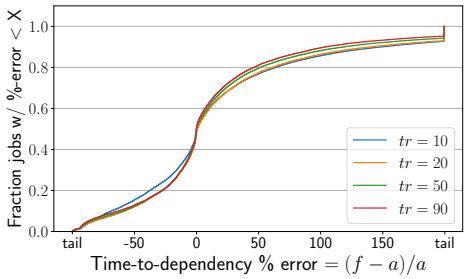


Figure 3: (T2) Time-to-dependency (TTD) prediction. This figure shows our predictor's performance on predicting TTD from the submission time of the upstream job, at different settings of tr in a CDF. f is the *forecasted* TTD, and a is the *actual* TTD. While being more precise ($tr \rightarrow 100$) does not yield better TTD predictions, it does affect predictions on whether or not a job will arrive.

recall for our model using various settings for the prediction threshold tr. As the model becomes more selective with respect to which downstream jobs will arrive ($tr \rightarrow 100\%$), it retains less relevant dependencies in total, but the dependent recurring jobs it predicts to arrive mostly do show up. The reverse is true as the model becomes less selective ($tr \rightarrow 0\%$).

We discuss the evaluation of our model based on a threshold that balances precision and recall. A common way to identify such a threhold is to select the threshold that maximizes *precision* $*$ *recall*. We find that $tr = 20\%$ yields the greatest *precision* $*$ *recall*, and therefore evaluate our model by setting $tr = 20\%$. The threshold used in an online prediction service can similarly be tuned from week-to-week based on observed precision and recall, though the specific target to optimize depends on the penalties associated with making mistakes in recall or precision.

(T2) metrics and model performance. To evaluate the performance of our model on predicting *when* a downstream job $j_d \in J_d$ will arrive at the arrival time of an upstream job j_u, j_d must satisfy two conditions: our model must predict j_d to arrive based on jobs that have already arrived during a point in time in the execution trace *and* it must actually arrive. Our evaluation focuses on jobs that satisfy both above conditions.

To evaluate the performance of our model for (T2), we use the *Root Mean Squared Error (RMSE)* and the *Median Absolute Error (MAE)* metrics to measure prediction error in absolute time units. RMSE measures error by computing the root of the average of squares of errors, while MAE measures error by computing the median of absolute error = $|forecast - actual|$, over all predictions. To measure relative error, we use the *percentage error* metric: it computes $(forecast - actual)/actual$ for each prediction.

While we discuss the evaluation of our model on (T2) setting $tr = 20\%$, we find that confidence in job arrival prediction only slightly affects time-to-dependency prediction quality. This does not mean that the setting of tr is inconsequential, as tr affects the predictions of whether or not a job will arrive. Here, we evaluate the time-to-dependency predictions only for jobs that are both predicted to arrive *and* actually arrive.

We observe the RMSE and MAE of our model to be 2.5 hours and 22 minutes, respectively: MAE is smaller, as RMSE can be skewed by large mis-predictions at the tail. While our absolute errors can be improved using more sophisticated techniques, we find that our model predictions are reasonable for most jobs in our workload in terms of relative error, as shown in Fig. 3 in the form of a cumulative distribution function (CDF), for different settings of tr: the arrival of 50% of arrived jobs $\in J_d$ are predicted within $\pm 20\%$ of its actual arrival. But, there is also a non-trivial number of significant over-estimates: the arrival of 7% of arrived jobs $\in J_d$ are over-estimated by 2× or more—i.e., the actual jobs arrive more than 2× earlier than predicted. While this may not explain all mis-estimations, we have found that aperiodic recurring jobs (such as those that are manually triggered) and jobs that depend on the outputs of multiple jobs are prone to greater mis-estimates (our simple model presented here only tries to predict the arrival of a future job based on one of its directly upstream recurring jobs).

4 The Wing dependency profiler

This section describes Wing, an end-to-end dependency profiler meant to be run intermittently (e.g., weekly) that uncovers historical, hidden inter-job dependencies from data provenance logs. It performs a series of analyses using these inter-job dependencies in-tandem with historical job telemetry, yielding characterizations of jobs and inter-job dependencies such as signs of misconfigured priorities between recurringly-dependent jobs (§2.3), predictability of upcoming jobs (§3), and estimates of recurring jobs' aggregate value considering their impact on downstream jobs that rely on their outputs, directly or indirectly (§4.3). These characterizations are ultimately used to inform better scheduling decisions, where its benefits are explored in more detail in §7.

4.1 Architecture

First, we introduce related systems and data sources upon which Wing depends, and provide an overview of Wing's architecture, shown in Fig. 4.

Input data sources. Wing relies on the following data sources, from which we derive job dependencies and insights thereof: (i) *JobRepo* preserves job telemetry (e.g., compute-hours, submission/completion time, and job structure metadata) for submitted jobs. Wing uses JobRepo to derive recurring jobs and their historic statistics. (ii) *ProvRepo* tracks data provenance across Microsoft to support auditing and compliance applications [56]. Specifically, it stores data provenance across systems deployed within Microsoft, including but not limited to Cosmos. ProvRepo is used by Wing to uncover historic inter-job dependencies, and from there, infer recurring dependencies between recurring jobs.

Analysis pipeline. Wing's data analysis pipeline, primarily composed of a workflow of inter-dependent SCOPE jobs, is managed by a workflow manager and is periodically executed in Cosmos. The pipeline reads data from JobRepo and ProvRepo and writes its output to be consumed by WingStore.

WingStore. The *WingStore* is a service that hosts the resulting analyses of Wing's analysis pipeline, periodically renewed each time a new instance of the pipeline completes. Given a historical job or the identifier of a recurring job, one can look up relevant historical job and inter-job dependency data: Such historical job data include, but are not limited to, distributions of job runtime and compute-time used. Historical inter-job data include distributions of job fan-in/fan-out, recurring inter-job dependencies, and distributions of number of downstream jobs. The WingStore is the interface between Wing's analysis and a Wing-guided resource manager.

4.2 The Wing pipeline: Single-hop analysis

Wing considers both *single-hop* and *multi-hop* dependencies in its analyses. The former occur when jobs directly consume the output(s) of another job. The latter are indirect dependencies between jobs that are connected by means of intermediate jobs. Here, we focus on the derivation of single-hop dependencies, which multi-hop dependencies are built upon, from historical provenance data stored in ProvRepo.

Single-hop dependency derivation. To derive single-hop dependencies from provenance data in ProvRepo (stored roughly in the form of <`input`, `operation`, `output`>, but with much more detailed context), we perform a self-join on the ProvRepo dataset with the condition of p_1`.input = `p_2`.output`. A näive self-join across multiple months of data is extremely compute intensive and can yield incorrect results, as a single file can be written multiple times by different jobs. To reduce join complexity and ensure correctness, we apply the following additional rules on the join:

(1) *R/W correctness*: The read must occur after the write. i.e., p_1`.operation` must occur after p_2`.operation`.
(2) *Last-writer wins*: If multiple writes occur on a single file, the read only depends on the latest write prior to the read.
(3) *Time windowing*: The time between the read and the write operations are at most T days, where we set $T = 30$.[7]

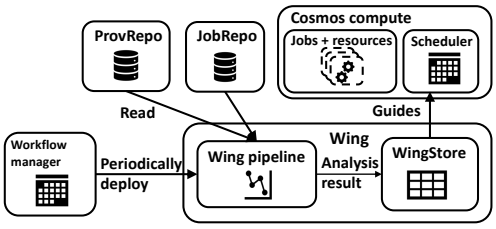


Figure 4: Wing architecture. A workflow manager periodically submits Wing's pipeline to Cosmos. Upon pipeline completion, results of its analyses are loaded in to WingStore, which informs Wing-guided schedulers (§6.2) with job and dependency characteristics.

Time windowing can reduce join complexity and allow our analyses to account for inter-job dependencies more *fairly*—if time windows are not applied over an observation period, operations issued earlier necessarily have a higher chance to be depended-upon. In other words, for each operation between days 31–60[8] in our dataset, time windows give them equal opportunity (in wall-clock time) to be depended-upon by directly-dependent operations.

Heuristics to identify recurring jobs & dependencies. A key to the analyses that we perform is the identification of recurring jobs, for which we employ the time-tested heuristic proposed in Morpheus [34] and applied in multiple production environments [34, 51]. Through the identification of recurring jobs and uncovered single-hop dependencies, the Wing pipeline further derives recurring dependencies and uncovers dependency characteristics of jobs using similar heuristics, described in Table 1. While ideally, we would like the full semantics of how inter-job dependencies are formed, due to the availability of the many different ways to submit a job (§2.1), our usage of heuristics is necessary. Sampling 25 jobs for manual verification, we confirm that our heuristics categorize jobs and dependencies correctly for 24 of the jobs.

4.3 Motivating multi-hop analysis: Job valuation using aggregate downloads

Companies can benefit more from their infrastructure investment through effective scheduling that prioritizes the completion of the most valuable jobs. But, often times, inter-job dependencies have not been considered when evaluating the importance of jobs—e.g., a job with high value can potentially depend on jobs with low value. In these cases, inter-job dependency awareness is key to ensure that upstream jobs do not disrupt high-value downstream jobs. Here, we look at why inter-job dependency analyses beyond direct dependencies (i.e., *multi-hop analyses*) can inform better, dependency-aware

[7] In retrospect, we should have set $T = 31$ to capture all monthly cycles, but our results based on $T = 30$ remain valid because (1) 98% of dependencies occur within a week, and (2) jobs of 89% of templates (97% of all jobs) have mean inter-arrival times of less than a week.

[8] Operations between days 31–60 are analyzed because we observe fully over time windows of 30 days both operations they depend on (days 1–30) and those that depend on them (days 61-90).

valuation of jobs to improve scheduling, and explore using the *number of downloads attained* associated with the outputs of a completed job as a proxy-metric for job value.

Priority assignments. To prioritize jobs today, schedulers in most production data analytics environments, including in Cosmos, use priority assignments to determine a job's order in its claim to resources. In this context, the notion of job value is often translated into a priority assignment on the job—the greater a job's value, the higher its priority. However, priorities in clusters are difficult to set correctly (Observation 2), and even at Microsoft, whose multi-billion dollar clusters are carefully provisioned and whose user-base is highly skilled, incidents triggered by late completion of hand-picked, closely monitored, and highly valued production jobs still occur due to mis-configured priorities.

Multi-hop value impact. The completion of a job can often be associated with some measurement of monetary value to a company. For example, jobs computing Bing's search indices directly impact the revenue of Microsoft. We term the direct value associated with the completion of a job its *job-local* value. However, the delay or failure of a job may not only affect its users and consumers of its output: through analyses of Cosmos's job DAG (Observations 3 and 5), we find that the delay or failure of certain jobs impact a lot more jobs and users than others. Hitches in the execution of these jobs are likely to cause much more financial and operational damage to users and organizations within the company due to the ripple effects they can create downstream, yet their impact might not always be obvious. While prior work [18, 34] suggest that finishing jobs prior to the arrival of their first directly-dependent job is important, quantifying the *aggregate value* of a job necessitates inter-job dependency analyses extending beyond a single hop (i.e., *multi-hop analyses*). Fig. 5 showcases a toy example that computes such an aggregate value for the root job of a dependency tree.

Approximating value impact with agg. downloads. Although determining the true dollar-value of jobs is difficult, we find it promising to evaluate the importance of jobs based on their historical *aggregate user downloads*, which measures hypothetically if a job fails, how many download operations it will affect (directly or indirectly) in total. In developing Wing, we have also experimented with several alternative metrics e.g., sum of cpu-hours and number of downstream jobs. Number of downloads was preferred by our resource management team because file downloads (1) are the most direct way users interact with a job's output; (2) can be easily interpreted and understood; and (3) because file downloads can be used to quantify how soon the output(s) of a job are used upon its completion. The properties of file downloads allow aggregate downloads counts to provide a proxy-measure to how the delayed or failed outputs of jobs can impact users in and out of Microsoft. Aggregate download counts also implicitly capture the number of downstream jobs that can be impacted by the failure of a job through their associated output downloads.

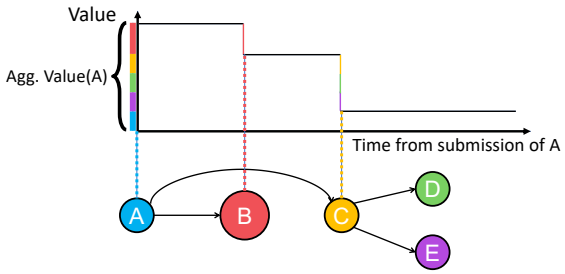


Figure 5: Value aggregation and value decay. In this toy example, jobs A–E are submitted at strict, absolute times, where the x-axis denotes time relative to the submission of job A. B and C have hard dependencies on A, and D and E have hard dependencies on C. The aggregate value of A is the sum of the aggregate values of B and C and A's own job-local value. With Wing, we can model how the aggregate value of A decays as it fails to complete by the time its downstream jobs arrive, losing the value of B at the time of B's submission, and collectively losing the values of C, D, and E at the time of C's submission (D and E depend *indirectly* on A through C, so if C fails, D and E will also fail). In this example, A retains its job-local value until the end.

While further work is required to confirm that aggregate download counts represents job value and to explore how it should be combined with other signals (e.g., user-provided priorities), we use it in this paper as our approximation of value.

Sanity-checking aggregate downloads as job value. We conducted a sanity check, using aggregate download counts for job valuation to see how it matches up with pre-existing notions of job importance. To that end, we obtained a list of six recurring job templates hand-curated by the Cosmos resource management team at Microsoft, each vetted to be significantly important to Microsoft's operation. We then look at Wing's ranks of those jobs.

Our results show that our valuation scheme mostly holds up for the most important jobs: We find that jobs of five of the six templates are consistently ranked by our scheme to be among the top 4% of all jobs submitted, with jobs of one template still ranking in the top 11%. We also measure relative rankings by user-specified priority and by our heuristic among jobs submitted to the same organizational queue, since priorities are only relevant when compared to other jobs sharing the same queue. For four out of the six hand-curated job templates, Our heuristic produces organizationally-relative rankings within 5% of priority assignment rankings. For one of the six job templates, our valuation scheme produces a ranking lower than that produced by priority assignments by up to 11%. For the last of the six job templates, however, we produce a ranking *higher* than that produced by priority assignments by 50%. This is surprising because we expected priority assignments for these six job templates, which are all verified to be highly important, to be extremely well-tuned, with highly-ranked priorities assigned to jobs of all six templates. Yet, jobs of the last template are only ranked at the 49^{th}

percentile of all submitted jobs within its queue by priority assignment—this mis-configuration may lead to significant issues once the queue becomes more heavily-loaded.

Future work: Further validating agg. downloads as value. We acknowledge that accurate job valuation is a difficult problem that requires further study, and that different companies can have different notions of job value. While further efforts are ongoing at Microsoft to validate the efficacy of our job valuation scheme (e.g., conducting surveys of Cosmos users), Cosmos's resource management team has noted that our valuation scheme is better than any of their existing heuristics used for job valuation, and are considering adopting it to aid in rolling out job upgrades and using it as a weighting function to report certain cluster performance indicators (e.g., reliability).

4.4 The Wing pipeline: Multi-hop analyses

Wing provides a flexible iterative solution implemented on top of SCOPE for performing *downstream multi-hop analyses*, in which for a given job, we analyze properties of its directly and indirectly-dependent jobs. Provided a set of single-hop inter-job dependencies, our framework allows the computation of both the transitive closure and aggregate statistics of all sub-DAGs rooted at each job in an inter-job dependency DAG (defined in Observation 5). Such multi-hop analyses are important to effectively guide scheduling decisions, as it can compactly characterize each job's downstream impact: i.e., if a job fails or is delayed, how will its downstream jobs and users be affected (§4.3)? Our framework generalizes the algorithm proposed in Owl [9], which allows multi-hop dependency analysis to be applied to other applications, e.g., fixing priority inversions[9] for Cosmos jobs.

Algorithm input. Our algorithm input is a single-hop job dependency DAG specified as a relational table, where the first column (`job`) holds the dependent job and the second column (`depOn`) holds the depended-upon job.

Algorithm output. Our algorithm outputs a relational table describing multi-hop dependencies. The first column (`job`) holds the downstream job, the second column (`depOn`) holds the (potentially multi-hop) upstream job, and the third column (`agg`) holds Wing-computed weights aggregated along all paths between the pair of up/downstream jobs.

Aggregation Functions (AFs). Each downstream multi-hop analysis specifies the following *Aggregation Functions* (AFs):

- *Weight function* (`wt_fn`): `wt_fn` takes in a job and its in- (or out-) edges as input, and outputs a weight `wt` for each graph edge. This operation is done once to convert the input DAG into an edge-weighted DAG.
- *Edge operation* (`e_op`): For two vertices t and v connected by an intermediate vertex u, `e_op` performs an aggregation of weights between a pair of (potentially auxiliary) in- (t,u) and out-edges (u,v) of u, constructing a new auxiliary weighted edge connecting t and v. Specifically, it computes the weight for an auxiliary edge based on new edges explored in each iteration between two indirectly connected jobs. This operation should be *distributive* over the `p_op` (defined following).
- *Path operation* (`p_op`): `p_op` aggregates weights on all explored paths between two jobs. While a unique path cannot be explored multiple times, the algorithm can make multiple traversals and aggregations between the same pair of up- and downstream jobs if multiple paths between two jobs exist. This operation should therefore be *associative*.
- *Downstream operation* (`ds_op`, optional): The downstream operation is the last step performed, after our iterative algorithm converges. For a job, it performs an aggregation on all of its downstream jobs and aggregated path weights.

Algorithm outline. We first preprocess the job dependency DAG with the AF `wt_fn` to generate the DAG edge weights `wt`. Then, for each job in parallel, our algorithm traverses the DAG and computes transitive closures along all paths, maintaining an "aggregated version" of `wt` using `e_op` and `p_op` along the way. Our algorithm completes in $O(log(diameter))$ iterations, where *diameter* is the longest path in the DAG. In each iteration, the algorithm maintains a *frontier* and a *base* table, both with the schema (`job, depOn, wt`). The frontier table records the set of discovered furthest reachable upstream jobs by `job` in `depOn`, while the base table records the set of all discovered reachable upstream jobs by `job` in `depOn`. The `wt` column of both tables records the aggregated weights along discovered paths from `job` to `depOn`. Each iteration joins and updates the frontier and base tables, extending the "reach" of each job by a maximum of 2×. Our algorithm pseudocode is shown in Algorithm 1.

4.5 Job value aggregation with Wing

4.5.1 Job value aggregation properties

Fair multi-hop time windowing. Aggregating value directly on even a single-hop time-windowed job dependency graph has a critical shortcoming: when considering multiple hops, jobs at the start of the observed trace still hold an advantage over jobs toward the end of the observation window in terms of opportunities to have their multi-hop downstream dependencies also land in the observation window. To better illustrate this, suppose we are given a recurring job template X with multiple jobs in our observation window. While ideally all jobs of X should have similar amounts of downstream dependencies, jobs of X that occur earlier in the trace are more likely to have their downstream dependencies also observed in the trace, while later jobs of X in the trace are more likely to have their downstream dependencies cut off due to the limits of using a static-length trace. In the limit of using an infinitely long trace, no time windowing is necessary.

A *multi-hop time window* is therefore needed to further

[9] Wing can fix priority inversions by raising the upstream job's priority before its dependent high-priority job arrives. Traditional OS methods require both jobs to have arrived at the scheduler, and dependency between the two jobs is communicated through concurrency data structures (e.g., locks). There is no lock-equivalent in Cosmos's scheduler.

```
// Helper functions
Function preprocess(s_hop) is
    gp_by_job = job G wts=wt_fn(depOn)(s_hop);
    return π job, depOn=wts.depOn, wt=wts.weights(gp_by_job);
end
Function extend_reach(t1, t2) is
    e_agg = π t1.job, t2.depOn, wt=e_op(t1.wt,t2.wt)(
                t1 ⋈ t1.depOn=t2.job t2);
    return job, depOn G p_op(wt)(e_agg);
end
// Computation start
Input : s_hop                          // Single-hop dependencies
i = 0;                                 // Iteration
ftr_i = preprocess(s_hop);             // Frontier
base = COPY(ftr_i) ;                   // Base
// base at the end of iter i covers deps up to 2^i hops
do
    i++;
    base_tmp = base − ftr_{i-1};
    ftr_i = extend_reach(ftr_{i-1}, ftr_{i-1});
    base_tmp = extend_reach(ftr_{i-1}, base_tmp) ∪ base;
    base = job, depOn G wt=p_op(wt) base_tmp;
    base = base ∪ ftr_i;
while COUNT(ftr_i) > 0;
return job, depOn G agg=p_op(wt) base;          // Converged
```

Algorithm 1: Multi-hop downstream analysis framework. `preprocess` first assigns weights to DAG edges with `wt_fn`. In each iteration, it calls `extend_reach` to further explore the graph from each job in parallel. In `extend_reach`, auxiliary edges with edge weights specified by `e_op` are created to denote newly discovered indirect dependencies (through the `JOIN`, or ⋈ operator). The auxiliary edges are deduplicated with a `GROUP BY` ($\mathcal{G}$) operator at the end of each iteration, yielding edge weights of `p_op(wt)`.

restrict the set of jobs eligible for value aggregation. Our multi-hop time windowing method works as follows: we first define a time window size ω smaller than the observation period. For each *valid* job j in the trace, we consider its entire set of directly and indirectly dependent jobs that are submitted by up to ω after its completion time. Here, we define valid jobs as jobs that complete at least ω prior to the end of the observation period. We set ω to *one week* for multi-hop dependency analysis, as the scale of the inter-job dependency graph bottlenecks transitive closure computation as ω increases: increasing ω exponentially increases the number of multi-hop inter-job dependencies to consider, as dependencies fan-out further into the future. ω is set to a week here to capture the majority of recurring dependencies that occur on a sub-weekly cadence (most recurring templates are submitted with inter-job arrival times of a day or less), while allowing our entire analyses pipeline to finish in approximately a day.

Value conservation. To conserve the total amount of value in the system, we employ an *equal contribution* scheme proposed in Owl [9], where each job contributes value to its directly-dependent upstream jobs equally, and the aggregate value of a job in this scheme is computed as the sum of value contributed upstream by all of its downstream jobs plus the value of the job itself. In this scheme, if a job j depends directly on the output of N jobs, it contributes $1/N$ of its value to each of its jobs directly upstream. Each of the N upstream jobs in turn further propagates j's (and their own) value upstream in the same fashion; e.g., if each of the N jobs directly depend on the output of M other jobs, j contributes $1/(N*M)$ of its value to each of the $N*M$ jobs two hops upstream. This yields the following equation, as proposed in Owl [9], for computing the aggregate value of a job:

$$agg_val(j) = \sum_{d \in \mathcal{D}_j} \left(\sum_{p \in \mathcal{P}_{(j,d)}} \prod_{e \in p} w_e * k_d \right) + k_j,$$

where $\mathcal{D}_j$ represents *all* downstream jobs of j, $\mathcal{P}_{(j,d)}$ represents all paths from j to d, w_e represents the weight of a directed edge e on the path p, and k_d and k_j represent the job-local values of d and j, respectively.

4.5.2 Wing value Aggregation Functions

We implement Owl's dependency-driven job valuation scheme with Wing's downstream multi-hop analysis framework, specifying Aggregate Functions as follows: the *weight function* `wt_fn` takes in a job j and its N upstream dependencies as input, and returns $1/N$ as the weight of each in-edge; the *edge operation* `e_op` multiplies the weights of its two operands; the *path operation* `p_op` sums the weights of its operands; and finally, for each job j, the *downstream operation* `ds_op` sums the job-local downloads of each job downstream of j multiplied by the aggregated path weights between the downstream job and j. j's job-local downloads are finally added to the downloads computed by `ds_op`, yielding j's aggregate downstream downloads.

Extensibility. While we elect to use downloads as a proxy for job value, Wing's framework is flexible enough to consider other metrics: e.g., if one day the dollar value associated with a job can be known, computing the aggregate downstream dollar value of a job is as easy as replacing a field in `ds_op`.

4.5.3 Aggregate value exploration and convergence

Using downloads as a proxy-metric for value, Fig. 6 shows the fraction of aggregate value explored in each iteration for each job on average. Considering the aggregate value of jobs with Wing allows us to uncover 83% of value that would otherwise be hidden if only job-local values (*iteration* = 0) were considered. In the context of value-based job scheduling (§5), this means that nearly 6× of value can be hidden from the scheduler if jobs are independently considered. The figure also shows that 99% of average job aggregate value can be explored within four iterations of our algorithm.

5 Wing-Agg: Inter-job value scheduling

Value scheduling. The objective in *value scheduling* is to maximize the value achieved from executing jobs in a workload, where the completion of each job is directly associated with an amount of *job-local value* attained. Job-local value can decay over time, and this behavior is often modeled as a *value function* (VF) in scheduling literature, which expresses value attained as a function of job completion time. In value

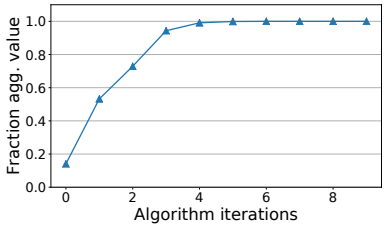


Figure 6: Aggregate value convergence. This figure shows the fraction of average aggregate job value uncovered downstream in each iteration of our value aggregation algorithm. 99% of aggregate value is discovered within four iterations.

scheduling, it is therefore important to complete jobs in a timely manner to achieve the most value.

Value and priority. We make a clear distinction between the terms *value* and *priority*. In this paper, we use the term *value* to describe a measure of "goodness" achieved associated with the completion of a job. *Priority*, on the other hand, defines the order in which pending jobs are assigned cluster resources: the higher the priority, the earlier a job receives its requested resources. Most commonly, including currently within Cosmos, the priority of a job is assigned by its submitter.

Wing-Agg. When inter-job dependencies are present, we find that it is important to consider the potential value downstream that can be lost if a job fails or is delayed. To consider the effects of inter-job dependencies, we propose a scheduling policy, *Wing-Agg*, that incorporates Wing's notion of inter-job dependencies into job priorities: *the goal of Wing-Agg is to achieve the most value for a given workload.*

As suggested in the introduction, completing the most value-impactful job may not lead to a scheduler attaining the most value, as some value-impactful jobs can also require large amounts cluster resource-time to complete. Indeed, prior work [8, 28, 44] has shown that schedulers can often benefit by considering together how much value a job provides and how much resource-time a job uses.[10]

Wing-Agg therefore considers the *aggregate value efficiency* of jobs, which measures how much aggregate value per aggregate resource-time a job impacts downstream. Essentially, Wing-Agg replaces user-assigned priorities with what Wing believes is a job's aggregate value efficiency. When a job arrives, Wing-Agg performs a look-up in the Wing-Store (§4.1). If the job is recurring, Wing-Agg computes the job's aggregate value efficiency by dividing the job's median historic aggregate value by its median historic aggregate compute-time, and assigns the quotient as the job's priority. If the job is ad-hoc, Wing-Agg estimates the job's aggregate value efficiency based on previous ad-hoc jobs that the same user has submitted. Wing-Agg assigns aggregate value efficiency rather than aggregate value as jobs' priorities to optimize for high value throughput.

[10] Although Wing-Agg and shortest-job-first both use job resource-time in their decisions, Wing-Agg frequently runs longer, more value-providing jobs ahead of shorter jobs.

6 Experimental setup

This section provides an overview of the Cosmos resource management infrastructure, describes our evaluated scheduling policies, and describes our experimental methodology.

Downloads attained as value. In our experiments, we use the number of downloads associated with the outputs of each job as a proxy for the value attained by a job. We model download attainment using real-world output download traces: if a job j completes at 1PM in the real-world (from the trace) but only completes at 2PM in our experiment, j attains only the output downloads associated with its outputs that occur after 2PM, and loses the downloads that occur between 1 and 2PM. A limitation of our model of value is that it does not reward completing a job early. Further research is required to determine how much additional value the early-completion of a job yields in data lakes.

Cosmos backend: YARN and hierarchical queues. Cosmos uses a *YARN-based resource manager* [12, 54] in the backend and utilizes *hierarchical queues* (queues, for brevity) to delineate resource boundaries between organizations—users/workflow managers can only submit SCOPE jobs to queues belonging to organizations of which they are a part. Cosmos uses a scheduling policy similar to the default policy that the *CapacityScheduler* in stock YARN uses, which orders jobs in each queue based on their (often user-) assigned priorities. A key difference is that jobs are scheduled with gang semantics in Cosmos—a job is admitted only when the scheduler can ensure that a user-provided minimum number of parallel, *job-requested* resources can be granted to it.

6.1 Simulation setup

We evaluate the application of Wing's analyses to scheduling using simulation-based experiments due to the scale of Cosmos: the Cosmos traces we use contain ~40k jobs per day, and ~160k inter-job dependencies. Experiments at this scale cannot realistically be attempted on research clusters without down-sampling jobs, at which point much inter-job dependency fidelity within the original workload will have been lost. We therefore use simulations to preserve the characteristics of inter-job dependencies in our experiments.

Simulation platform: design and implementation. Our simulation platform takes a discrete-event based approach. To ensure that our experiments retain most properties of YARN/Cosmos, our simulation platform makes minimal changes to the YARN architecture—our implementation only mocks out the real-time clock and the communication layers of the YARN servers. We also use real queue sizes for each hierarchical queue in our Cosmos cluster. The authors plan to contribute this simulator back to the open source community.

Simulation accuracy. To make simulation feasible given the scale of our job logs, the simulator does not model: (1) "internal" dependencies among stages of a job, but rather treat a job as a rigid collection of tasks; (2) resource-sharing through opportunistic execution [35] of job tasks, which allows jobs

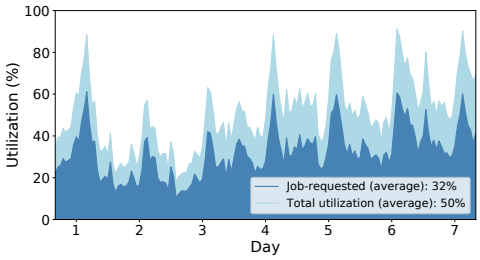


Figure 7: Cluster utilization. This figure shows the job-requested and total resource utilizations of our real cluster.

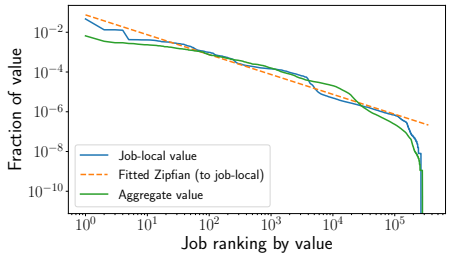


Figure 8: Distribution of job value. This figure shows the distributions of job-local value and aggregate job value, along with a Zipfian distribution fitted to job-local value. The distribution of job value deviates from Zipfian at lower job rankings.

to use more resources than requested when those resources are otherwise idle; and (3) job sizes based on resources used rather than job-requested resources, meaning that our simulations only consider the deep blue area in Fig. 7.

To evaluate the fidelity of our simulator, we measure the absolute differences in job completion times between jobs in our simulations (using the baseline system policies) and the same jobs run in the real cluster. We normalize the deltas by the job's real-world latency, and observe that even at the 99^{th} percentile, jobs are shifted by only 1.3% of their latencies. Our experiments run at 100% cluster capacity also achieve average resource utilization for job-requested resources within 1.5% of what is observed in the real cluster.

6.2 Evaluated scheduling policies

In addition to Wing-Agg (§5), we evaluate value-attainment on our workload traces on the following scheduling policies. All implement Cosmos's gang-scheduling semantics.

PRIO represents Cosmos's current approach, and is the default scheduling policy used by stock YARN in its CapacityScheduler. It orders jobs within each hierarchical queue based on user-specified priorities.

Wing-MIL. Millennium [8] is a VF-aware scheduler that orders jobs based on expected value attained per resource time: For each queued job it computes how much value can be gained at an estimated job completion time, divides the value by total job resource-time, and orders jobs by the resulting quotient. MIL is our implementation of Millennium on YARN, following descriptions in its design as closely as possible.

Wing-MIL is MIL using Wing-informed value functions (VFs): In addition to capturing how the job-local value of a single job decays, a Wing-informed VF captures *potential* value associated with the job lost over time by modeling a job's full decay of downloads. A job j attains all of j's aggregate downloads in the most optimistic case if it completes before or at its real-world completion time; otherwise, it loses value according to when users perform download operations *and* when downstream jobs fail due to it not completing on time (illustrated earlier in Fig. 5). For example, in a Wing-guided VF, if j completes at 1PM in the real-world but only completes at 2PM in our experiment, j loses all the direct downloads that occur between 1 and 2PM, *and* all the *indirect downloads* rooted in jobs that directly depend on j submitted between 1 and 2PM.

Plan-ahead based VF-aware policies. We attempted to evaluate more sophisticated plan-ahead based VF-aware policies, e.g., FirstOpportunityRate [44]. But, we found that one implementation of such a policy couldn't accommodate workloads at Cosmos scale, and efforts to mitigate bottlenecks by caching and limiting plan-ahead led to less value attainment than simpler policies (e.g., MIL). We therefore do not include our attempts with such a policy, as further work is warranted before conclusions are drawn.

6.3 Workload and predictor descriptions

Dataset. We use data from the final four weeks of our analysis dataset to evaluate our scheduling policies: Within the four weeks of data, Wing uses data in the first and second weeks to establish job and dependency profiles. Experiments are conducted over the third week, and downloads (value) are counted for each job up to one week (into the fourth week) from the completion of the job. Each day of traces contains ~40k jobs and ~160k inter-job dependencies.

Considering inter-job dependencies. Different from prior work, our experiments take characteristics of inter-job dependencies into account to realize more realistic workloads. For example, if a job holds a hard dependency on the output of an upstream job but the output is not available in time, the job fails due to missing input. Other dependency patterns, such as polling behavior (when a job waits for its inputs to become available), are also modeled faithfully. Jobs and dependencies considered in our experiments are described in Table 1.

Job value distribution. Job value, as measured by the number of downloads associated with the timely completion of a job in our experiments, are distributed roughly in a Zipfian fashion ($s = 1$) with deviation at the low end, as shown in Fig. 8. This means that the most valuable jobs are downloaded significantly more times than less valuable jobs. When scheduling for value on a workload that is inter-job dependency aware, schedulers should work to *unblock* the most valuable jobs before they arrive in order to attain their value.

Value efficiency predictor. Wing-Agg and Wing-MIL use a predictor to estimate the aggregate value efficiency associated with upcoming recurring jobs to optimize for value throughput. While §3 shows that direct inter-job dependencies can be predictable, it neither considers predictions on a

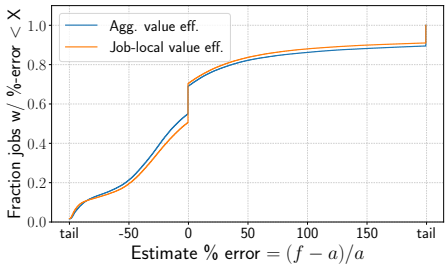


Figure 9: Value efficiency prediction. This figure shows the CDF of our predictor's performance on predicting the value efficiency and aggregate value efficiency of recurring jobs.

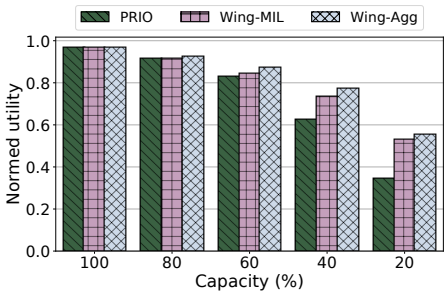


Figure 10: Benefits of Wing guidance. This figure shows the value attained for each scheduling policy, normalized to total value achievable. Wing-guidance (exemplified in Wing-Agg and Wing-MIL) is significantly beneficial at constrained capacities.

job's subgraph of downstream dependencies, nor a job's value impact. Evaluating predictions on aggregate value efficiency therefore allows us to better understand the performance of Wing-guided schedulers. For recurring jobs in our experiments, we use a median-based predictor to predict the value efficiency associated with a job. That is, given a recurring job j of template τ, we predict j's value efficiency based on the historical median value efficiency for jobs of template τ.

Fig. 9 shows the performance of our value efficiency predictor in a CDF. For predicting the aggregate value efficiency of a job, 39% of our predictions fall within ±20% of the actual value efficiency of a job, while for predicting the value efficiency of a single job, 44% of our predictions fall within ±20% of the actual value efficiency of a job. While we are working on further studies to improve predictor accuracy with more sophisticated methods, we find that the performance of our simple predictor enables Wing-Agg to outperform other evaluated scheduling policies in value attainment (§7).

7 Experimental results

We evaluate the efficacy of each scheduling policy for the actual full Cosmos resource capacity (100%) and for smaller capacities (at 80–20%). Value-attainment results are reported as a percentage of value achievable—i.e., if all jobs in workloads complete before any of their values are lost.

Cluster capacities & consequential policy decisions. Scheduling is most interesting when cluster capacity is constrained and schedulers need to make difficult decisions regarding which jobs to provide resources. Indeed, at 100% capacity, the baseline and more advanced schedulers perform similarly, completing > 99% of all jobs in the trace. We find that the lower cluster capacities (i.e., ≤ 40%) best exemplify the consequences of decisions a scheduler makes. We therefore focus the discussion of our results at these capacities to maximize observable differences.

Takeaways. Our experiments yield the following key takeaways. First, policies guided by Wing are better at achieving value when clusters are heavily-constrained. In particular, Wing-Agg outperforms all other compared policies at all capacities and improves value attained by up to 21% as capacity declines. Second, understanding the downstream impact of a job is crucial in constrained clusters, and that Wing-guided inter-job dependency predictions are accurate enough to be practical: Wing-Agg can effectively complete the prerequisites of the most consequential jobs. Finally, we demonstrate significant opportunity in applying inter-job dependency awareness in Wing to a cluster-wide queue and establishing a cluster-wide value metric: Wing-Agg achieves up to 93% of all value in our workload when using a single cluster-wide queue, using only 20% of cluster capacity.

7.1 Benefits of Wing guidance

Fig. 10 shows that policies guided by Wing beat PRIO at all capacities, with value attainment gaps widening as the cluster is increasingly stressed. At 60% capacity, Wing-Agg achieves 87% of value (vs PRIO's 80%). At 40% capacity, Wing-Agg achieves 77% of value (vs PRIO's 62%). Even at 20% capacity, Wing-Agg is able to capture more than half of all value (55%), while PRIO only captures 35% of value.

Considering aggregate value gives Wing-guided schedulers a two-fold benefit over PRIO. First, it naturally "fixes" priority mis-configurations, such as priority inversions, by propagating job value upstream, such that downstream jobs with high value are not blocked. Second, it guides schedulers toward sub-DAGs of high value efficiency jobs effectively, allowing schedulers to achieve more value with less resources.

Are ad-hoc jobs disadvantaged? Since Wing-Agg focuses on recurring jobs, we examine our logs to see if ad-hoc jobs are at a disadvantage when scheduled by Wing-Agg vs recurring jobs, where the priority of ad-hoc jobs are determined by the median aggregate value efficiency of previous jobs submitted by the same user. We find, from results at 20% cluster capacity, that 25% of recurring jobs fail, compared to 42% of ad-hoc jobs. However, recurring jobs also carry 9× more value than ad-hoc jobs. To optimize for value, Wing-Agg necessarily needs to complete larger fractions of recurring jobs. Indeed, recurring jobs are more often production jobs [34].

Dynamic priorities (Wing-MIL). Intuitively, policies using dynamic priorities (e.g., value functions, or VFs) such as Wing-MIL should perform better than static policies such as Wing-Agg, as VFs can express both importance and urgency while priorities only allow the expression of one of the two dimensions; but, we observe that Wing-Agg outperforms

Wing-MIL at all capacities, albeit only slightly.

Unlike Wing, which only depends on aggregate value-efficiency predictions, Wing-MIL also depends on the time-to-dependency predictions of directly-dependent jobs (§3) to determine when aggregate job value decays. But, while a part of this underperformance is indeed caused by imperfect predictions of time-to-dependencies, we find that providing Wing-MIL with perfect job value and time-to-dependency information does not help much. Further analyzing our results, we find that this underperformance is mainly due to Wing-MIL's failure to consider the *properties* of inter-job dependencies. For example, a downstream job that polls for the arrival of its inputs will not fail if its upstream jobs complete late. But, VFs constructed from historical data will still reflect a drop in value at the time the polling downstream job is expected to arrive, leading Wing-MIL to believe that it should give up prematurely on scheduling the job. This shortcoming can be addressed by considering dependency properties *explicitly*, but our attempted implementation of such a policy does not significantly improve over Wing-Agg: both Wing-Agg and our attempted implementation can complete the most impactful, value-efficient jobs in a timely manner.

Practicality of Wing-Agg. The simplicity of Wing-Agg is desirable from an engineering standpoint, as Wing-Agg is both highly practical and highly scalable: Integrating Wing-Agg into a production cluster requires minimal changes to the existing resource management framework, and all the information needed for Wing-Agg to determine a job's priority can be precomputed offline in Wing's analysis pipeline (§4). Adoption of Wing-Agg into production can therefore be straightforward, upon confirming job valuation schemes.

7.2 Sensitivity and ablation studies

Aggregate vs. job-local value. This section discusses benefits of understanding job value at an aggregate vs job-local level by comparing Wing-Agg against Wing-Direct, where *Wing-Direct considers the job-local value efficiency of a job*: i.e., Wing-Direct only considers direct-downloads associated with the outputs of and the compute-time of a single job only.

The patterned bars in Fig. 11 show the normalized value attained by Wing-Agg and Wing-Direct. While Wing-Direct outperforms PRIO, Wing-Agg maintains significant benefit over Wing-Direct at the tightest capacities: Wing-Agg attains 13% more overall value than Wing-Direct at 20% capacity. Our analysis finds that Wing-Direct's knowledge of job resource consumption allows it to effectively complete jobs at the head of queue, enabling it to complete a similar amount of jobs as Wing-Agg. But, with knowledge of historical aggregate value efficiency, we find that Wing-Agg completes jobs in the more value-heavy sub-DAGs of the inter-job dependency DAG, yielding significant improvements over Wing-Direct.

Wing predictions vs. Oracle knowledge. We examine how much potential benefit better predictions can provide to each Wing-aided policy. *Oracle Wing-Agg* represents Wing-Agg endowed with perfect knowledge of aggregate value efficiency, and *Oracle Wing-Direct* represents Wing-Direct provided with perfect knowledge of job-local value efficiency.

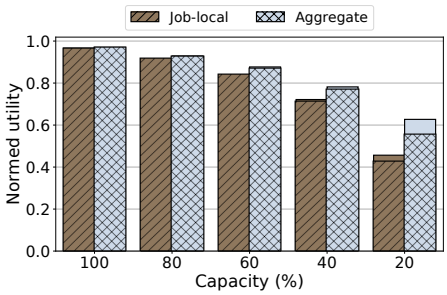


Figure 11: Benefits of aggregate job value. *Aggregate* (corresponding to aggregate download-aware) vs *Job-local* (corresponding to direct download aware only) bars show the benefits of aggregate value, compared to only scheduling based on job-local value. The solid portion of the bars show the benefits of Oracle knowledge.

While we find that having better predictions are beneficial, the differences between the solid (representing policies with Oracle knowledge) and the patterned bars (representing policies with Wing-provided predictions) in Fig. 10 and Fig. 11 show that at most capacities, Wing-guided schedulers achieve close to the value attained by their Oracle variants. However, having more accurate information presents opportunity for significant gain in value attained for Wing-Agg at 20% capacity: e.g., Oracle Wing-Agg improves value realized over Wing-Agg by 8% of overall value. Conversely, although Oracle Wing-Direct is granted exact knowledge of how value-efficient each job is, its view of the overall inter-job dependency graph leads to only incremental benefits.

Oracle benefits to aggregate value aware policies come from a more accurate knowledge of a summarized view of the inter-job dependency graph: compared to single job value-aware policies with Oracle knowledge, a policy such as Oracle Wing-Agg can efficiently complete the most consequential jobs in the job dependency graph, increasing value attained (by up to 18% of overall value vs Oracle Wing-Direct) and reducing the number of jobs failed due to missing input (by 3% of all jobs vs Oracle Wing-Direct).

Sensitivity to mis-predictions. We examine the sensitivity of Wing-Agg to aggregate value efficiency mis-predictions on our workload by running experiments that introduce artificial shifts in aggregate value efficiency provided by Oracle Wing, using 20% cluster capacity. Each experimental run is associated with a maximum artificial shift s, where $s \in \{1.1, 1.25, 1.5, 2, 5, 10\}$. For each job j within each run, we scale the aggregate value efficiency e_{val} of j provided by Oracle Wing by multiplying e_{val} by a randomly sampled multiplier m between $1/s$ and s. Our results show that Wing-Agg is not sensitive to mis-predictions in value on our workload: For $s \leq 5$, value attained is only reduced by at most 4% vs Oracle Wing-Agg. For $s = 10$, value attained is only reduced by 11%. This insensitivity is because job values in our workload are distributed in a Zipfian fashion (§6.3), where the most

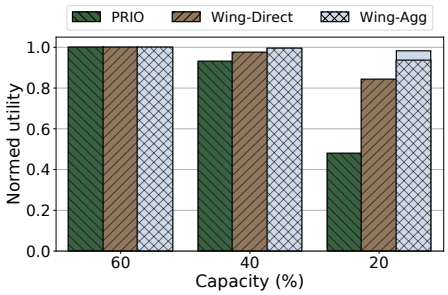


Figure 12: Benefits of Wing-guidance with a cluster-wide queue. This figure shows the value attained for policies from 60–20% cluster capacities in a cluster with a merged cluster-wide queue. All policies complete all jobs at 60% capacity. Wing-guidance (exemplified by Wing-Agg) is increasingly beneficial at lower capacities. The solid portion of the bars show the benefits of Oracle knowledge.

valuable jobs are much more valuable than other jobs.

Reducing transitive closure computation. At 20% capacity, Wing-Direct (0 iterations of Wing's multi-hop analysis) attains 42% of all value, while Wing-Agg (9 iterations executed) attains 55% of all value. In Fig. 6 in §4.5.3, we find that 99% of aggregate value of most jobs can be explored in four iterations of Wing's multi-hop analysis. We therefore believe that four iterations of exploration would be sufficient to similarly attain 55% of all value, and that two iterations of exploration would allow us to attain close to 50% of all value.

7.3 Cluster-wide queue and value metrics

Our earlier results correspond to a simplified view of Cosmos using strictly enforced queue boundaries. Hard queue boundaries restrict placement more than in the real system, where resource-sharing (§6.1) softens queue boundaries, which might exaggerate Wing-Agg's benefits. To confirm that Wing-Agg's improvements are not due to hard queue boundaries, we evaluate a boundary-free alternative with experiments run using a single global, cluster-wide queue.

Evaluation. Fig. 12 shows the value attainment of our evaluated scheduling policies using a single cluster-wide-queue. We note that all jobs are able to complete for all scheduling policies at 60% cluster capacity. Indeed, the dark blue area in Fig. 7 show that these requests peak at around 60%. At 40% capacity, the cluster still has more capacity than needed most of the time: Wing-Agg achieves 99% of value, and Wing-Direct and PRIO achieve 97 and 93% of value, respectively.

Under extreme capacity crunch (e.g., 20% capacity), removing restrictions of hard queue boundaries improves value attained of all policies. But, a Wing-guided scheduler sees significantly more benefit in terms of absolute value achieved. With a cluster-wide queue at 20% capacity, Wing-Agg attains 93% of value, whereas Wing-Direct attains 84%, and PRIO only attains 47%. Furthermore, Wing-Agg fails fewer jobs compared to both Wing-Direct and PRIO (11% vs 13% and 25% of jobs, respectively).

We find that understanding inter-job dependencies is critical, as Wing-Direct with Oracle knowledge did not significantly outperform Wing-Direct with predicted values, both in terms of value attained and in terms of number of jobs failed; yet, we find that Wing-Agg with Oracle knowledge, in this setting, can achieve up to 98% of all value (comparable to performances at 100% capacity), while failing only 7% of all jobs (compared to 26% in a multi-queued setting at 20% capacity). One of the reasons why Wing-Agg is able to attain 93% of all value using only 20% of cluster capacity is due to its ability "unblock" the most valuable downstream jobs.

Recall that the simulated job sizes in our experiments are based on job-requested resources, rather than job-used resources, which may be higher because of opportunistic execution. As a result, cluster utilization is lower in our experiments. But, we believe that the rankings of the different schedulers are not affected, because the number of opportunistic resources highly correlate with that of allocated job-requested resources, both across the top 10% of most valuable jobs (Spearman correlation of 0.85) and across all jobs (Spearman correlation of 0.84). Indeed, the amount of opportunistic resources available to a job is capped with a max proportional to the number of allocated job-requested resources [49]. So, the relative differences shown for 20% cluster capacity may instead be for 30% cluster capacity in the heavier workload.

Toward establishing a cluster-wide value metric. Our results confirm that removing queue boundaries would be beneficial. Partitioning resources into queues naturally introduces resource fragmentation, but usage of queues is often viewed as a "necessary evil," as certain organizations are willing to pay more to have guaranteed access to their share of compute. Yet, näively removing queue boundaries without a quota-system [55] in place may introduce resource competition, where users across different organizations assign increasingly high priorities to their jobs to acquire guaranteed resources. A cluster-wide, automated arbitrator that understands both system-internal (e.g., aware of downstream number of affected jobs and user-downloads) *and* organizational/user-defined notions of importance is therefore required. We see this as an exciting direction for further research.

Current state of deployment. Instead of immediately deploying Wing-Agg as described, the Microsoft Cosmos resource management team has asked us first to deploy an inter-job dependency advisory tool using analyses from Wing, to aid users on better configuring their jobs. The tool will allow us to gather user feedback on our recommendations.

8 Related work

Workflow managers. Workflow management for batch analytics jobs is a widely studied area in the fields of databases and data management [30, 40, 41]. Our work differs in two primary ways: (1) workflow managers often assume the availability of a dependency graph up-front, while Wing infers properties of inter-job dependencies from job history; and (2) workflow managers optimize only a single pipeline of jobs submitted by one user at a time, while Wing considers inter-

dependent jobs across workflow and organization boundaries.

Cluster workload analysis. Although much work has been done on cluster workload analysis from many different perspectives (e.g., resource/workload heterogeneity [1, 11, 26, 37, 45], failure analysis [7, 17, 46], job predictability [43, 50, 52], and intra-job task dependency [23, 24, 51]), most prior work assumes (implicitly or explicitly) that each job is independent of other jobs. This paper fills the knowledge gap with analyses of inter-job dependencies and application of this knowledge in cluster scheduling.

Cluster scheduling. Although a variety of work has been published in the area of cluster scheduling, each trying to address scheduling woes of different kinds of workloads (e.g., support for general batch analytics [5, 10, 18, 21, 22, 29, 34, 43, 53, 54], low latency scheduling [15, 16, 35, 42], and strategies to handle mixes of workloads [12, 19, 20, 48, 55]), most work in cluster scheduling similarly assume the independence of jobs. Our work shows that incorporating knowledge of inter-job dependencies can improve cluster scheduling in an environment with a lot of data and work product sharing, and we believe that considering inter-job dependencies can help future schedulers better tackle challenges, such as enabling better job task placement and learning better scheduling policies [38, 47].

Task-DAG schedulers assign resources to inter-dependent tasks within a job based on knowledge of the overall task-DAG [18, 23, 24, 38]. Such techniques and our proposed policies can be complementary, as task-DAG schedulers drill into job-level details while our schedulers (e.g., Wing-Agg) work at a higher level and treat jobs as black boxes. In particular, schedulers that predict the arrival of future jobs [34, 38] can benefit from the availability of inter-job dependency context to refine their predictions. Some task-DAG scheduling techniques could also be applied to the problem of inter-job dependency scheduling; but, these task-DAG schedulers generally assume upfront availability of task-DAGs, while full inter-job dependency graphs are rarely available ahead of time. An interesting direction for future research is in combining task-DAG scheduling techniques with some form of Wing-provided "probabilistic inter-job dependency" DAGs.

Jockey and Morpheus. Jockey [18] uses the direct dependencies of jobs to illustrate the importance of maintaining low job latency variance, but uses a step-function with value=1 until the *user-provided deadline* as each job's value function (VF). Morpheus [34] improves upon Jockey's notion of VFs by deriving deadlines based on a job's first consumer (as observed from historical instances of that job), but still considers all jobs as equal in value. In addition to our characterization of inter-job dependencies in a large analytics cluster, our work extends Morpheus and Jockey in two ways: (1) jobs no longer all have the same value—instead, Wing derives each job's value (and therefrom priority) as the sum of a chosen value metric (e.g., downloads) for all downstream dependencies, and (2) value is no longer a step-function with a single deadline based on a job's first direct consumer, but a rich decay proportional to the aggregate value of dependency sub-DAGs rooted in each direct consumer. While we do not directly compare against Morpheus, in §7.1, we find, in the context of Wing-MIL, that a premature drop in aggregate value can lead to the scheduler giving up early when dependency properties are not considered, leading to lower value attainment. Considering value as a step-function with a single deadline can therefore potentially be detrimental when inter-job dependencies are present in cluster workloads. While Wing-Agg uses only the initial "height" of the aggregate value VF of each job to set priorities, we believe that full aggregate value VFs can still better guide other scheduling decisions, such as determining which jobs to load-shift.

Systems using job recurrence and data provenance. There has also been much prior work on systems that efficiently collect provenance data [13, 39] and systems that both exploit job recurrence and data provenance on other problems [25, 33], such as garbage-collecting shared computation results. Our work uses similar ideas, but focuses on facilitating better value attainment in resource scheduling.

Owl and Guider. Our previous work Owl [9] and Guider [39] introduced the usage of job dependencies to determine the value of jobs. Wing operationalizes and expands upon prior work by (1) analyzing and characterizing inter-job dependencies in a large cluster, (2) evaluating predictability of recurring inter-job dependencies, (3) integrating inter-job dependencies into cluster schedulers, (4) applying said schedulers to a real scheduling problem, and (5) providing a general aggregate inter-job dependency analysis framework.

9 Conclusion

Complex inter-job dependencies pervade modern data lakes, creating complex problems as cluster schedulers make decisions without knowing of them. The Wing dependency profiler uncovers these dependencies from provenance logs and provides improved guidance to cluster schedulers. Evaluations with real job traces show that significantly more value, in terms of successful user downloads, can be attained by using Wing-guided priority assignments over those provided by users. Wing's effectiveness opens a new range of resource management possibilities guided by automatically-determined knowledge of the impact of jobs.

Acknowledgements

We thank John Wilkes (our shepherd) and our OSDI 2020 reviewers for their valuable feedback and suggestions. We also thank Raghu Ramakrishnan, Boris Asipov, Hiren Patel, Yiwen Zhu, Isha Tarte, and Panagiotis Garefalakis for their help throughout the development of this project. We thank the members and companies of the PDL Consortium (Alibaba, Amazon, Datrium, Facebook, Google, HPE, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure, Salesforce, Samsung, Seagate, Two Sigma, and Western Digital) and VMware for their interest, insights, feedback, and support.

References

[1] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18. USENIX Association, 2018.

[2] Anton Beloglazov and Rajkumar Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10. ACM, 2010.

[3] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation : Practice and Experience*, 24(13), September 2012.

[4] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, CIDR '15, January 2015.

[5] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[6] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2), August 2008.

[7] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *Proceedings of the 25th International Symposium on Software Reliability Engineering*, ISSRE '14. IEEE Computer Society, Nov 2014.

[8] Brent N. Chun and David E. Culler. User-Centric Performance Analysis of Market-Based Cluster Batch Schedulers. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02. IEEE Computer Society, May 2002.

[9] Andrew Chung, Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Panagiotis Garefalakis, and Gregory R. Ganger. Peering Through the Dark: An Owl's View of Inter-job Dependencies and Jobs' Impact in Shared Clusters. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19. ACM, 2019.

[10] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware Container Scheduling in the Public Cloud. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018.

[11] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[12] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19. USENIX Association, February 2019.

[13] Sergio Manuel Serra da Cruz, Patricia M. Barros, Paulo Mascarello Bisch, Maria Luiza Machado Campos, and Marta Mattoso. Provenance Services for Distributed Workflows. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08. IEEE Computer Society, 2008.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation*, OSDI '04. USENIX Association, 2004.

[15] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018.

[16] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15. USENIX Association, 2015.

[17] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations. In *Proceedings of*

the IEEE 37th International Conference on Distributed Computing Systems, ICDCS '17. IEEE Computer Society, June 2017.

[18] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12. ACM, 2012.

[19] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[20] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the 13th European Conference on Computer Systems*, EuroSys '18. ACM, 2018.

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11. USENIX Association, 2011.

[22] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016.

[23] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-resource Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016.

[24] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and Dependency-aware Scheduling for Data-parallel Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016.

[25] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI '10, 2010.

[26] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of the 2019 International Symposium on Quality of Service*, IWQoS '19. ACM, 2019.

[27] Alon Halevy, Flip Korn, Natalya F. Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing Google's Datasets. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16. ACM, 2016.

[28] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04. IEEE Computer Society, June 2004.

[29] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. ACM, 2009.

[30] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, March 2011.

[31] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting Subexpressions to Materialize at Datacenter Scale. *Proceedings of the VLDB Endowment*, 11(7):800–812, March 2018.

[32] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. Peregrine: Workload Optimization for Cloud Query Engines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[33] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18. ACM, 2018.

[34] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, November 2016.

[35] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15. USENIX Association, 2015.

[36] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD '09. IEEE Computer Society, Sep. 2009.

[37] Qixiao Liu and Zhibin Yu. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: A View from Alibaba Trace. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018.

[38] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the 2019 ACM Special Interest Group on Data Communication*, SIGCOMM '19. ACM, 2019.

[39] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Phil Cudre-Mauroux. Dependency-Driven Analytics: a Compass for Uncharted Data Oceans. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, January 2017.

[40] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10. ACM, 2010.

[41] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of MapReduce pipelines. In *Proceedings of the IEEE 26th International Conference on Data Engineering*, ICDE '10. IEEE Computer Society, March 2010.

[42] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13. ACM, 2013.

[43] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the 13th European Conference on Computer Systems*, EuroSys '18. ACM, 2018.

[44] Florentina I. Popovici and John Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05. IEEE Computer Society, 2005.

[45] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12. ACM, 2012.

[46] Andrea Rosà, Lydia Y. Chen, and Walter Binder. Predicting and mitigating jobs failures in big data clusters. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '15. IEEE Computer Society, 2015.

[47] Malte Schwarzkopf and Peter Bailis. Research for Practice: Cluster Scheduling for Datacenters. *Communications of the ACM*, 61(5):50–53, April 2018.

[48] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13. ACM, 2013.

[49] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. Autotoken: Predicting peak parallelism for big data analytics at microsoft. *Proceedings of the VLDB Endowment*, 13(12):3326–3339, August 2020.

[50] Liqun Shao, Yiwen Zhu, Siqi Liu, Abhiram Eswaran, Kristin Lieber, Janhavi Mahajan, Minsoo Thigpen, Sudhir Darbha, Subru Krishnan, Soundar Srinivasan, and et al. Griffon: Reasoning about Job Anomalies with Unlabeled Data in Cloud-Based Platforms. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[51] Huangshi Tian, Yunchuan Zheng, and Wei Wang. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[52] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A Kozuch, and Gregory R Ganger. JamaisVu: Robust scheduling with auto-estimated job runtimes. Technical report, Technical Report CMU-PDL-16-104. Carnegie Mellon University, 2016.

[53] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys '16. ACM, 2016.

[54] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13. ACM, 2013.

[55] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, EuroSys '15. ACM, 2015.

[56] Paul Voigt and Axel von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, San Jose, CA, 2012. USENIX Association.



RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers

Hang Zhu
Johns Hopkins University

Kostis Kaffes
Stanford University

Zixu Chen
Johns Hopkins University

Zhenming Liu
College of William and Mary

Christos Kozyrakis
Stanford University

Ion Stoica
UC Berkeley

Xin Jin
Johns Hopkins University

Abstract

Low-latency online services have strict Service Level Objectives (SLOs) that require datacenter systems to support high throughput at microsecond-scale tail latency. Dataplane operating systems have been designed to *scale up* multi-core servers with minimal overhead for such SLOs. However, as application demands continue to increase, scaling up is not enough, and serving larger demands requires these systems to scale out to multiple servers in a rack.

We present RackSched, the first rack-level microsecond-scale scheduler that provides the abstraction of a rack-scale computer (i.e., a huge server with hundreds to thousands of cores) to an external service with network-system co-design. The core of RackSched is a two-layer scheduling framework that integrates inter-server scheduling in the top-of-rack (ToR) switch with intra-server scheduling in each server. We use a combination of analytical results and simulations to show that it provides near-optimal performance as centralized scheduling policies, and is robust for both low-dispersion and high-dispersion workloads. We design a custom switch data plane for the inter-server scheduler, which realizes power-of-k-choices, ensures request affinity, and tracks server loads accurately and efficiently. We implement a RackSched prototype on a cluster of commodity servers connected by a Barefoot Tofino switch. End-to-end experiments on a twelve-server testbed show that RackSched improves the throughput by up to 1.44×, and scales out the throughput near linearly, while maintaining the same tail latency as one server until the system is saturated.

1 Introduction

Online services such as search, social networking and e-commerce have strict end-to-end user-facing Service Level Objectives (SLOs) [12, 22]. To meet such SLOs, datacenter systems behind these services are expected to provide high throughput with low *tail latency* in the range of tens to hundreds of *microseconds* [12]. Example systems include key-value stores [6, 7, 60], transactional databases [9, 64], search ranking and sorting [13], microservices and function-as-a-service frameworks [17], and graph stores [43, 67].

With the end of Moore's law and Dennard's scaling, applications can no longer rely on single-threaded code to execute faster on newer processors with increased clock rates and instruction-level parallelism [31]. This leads to the rise of multi-core machines to scale up computation. Meeting microsecond-scale tail latency is challenging given that the request processing times with single-threaded code on bare metal hardware are already in the same time scale. This means that the operating system (OS) should impose minimal overhead when it manages resources and scales up these applications on multi-core machines.

This calls for new software architectures to efficiently utilize the resources of multi-core machines. One critical component of such architectures is scheduling. Dataplane operating systems have been designed to support low-latency applications with minimal overhead to meet strict SLOs [14, 39, 54, 58, 59]. For example, Shinjuku [39] uses efficient mechanisms to implement preemption and context switching, in order to realize *centralized* scheduling policies to avoid head-of-line blocking and address temporal load imbalance between multiple cores.

However, as application demands continue to increase, *scaling up* a single server is not enough. Serving larger demands requires these systems to *scale out* to multiple servers in a rack, which is termed as rack-scale computers, such as Berkeley Firebox [1], Intel Rack Scale Architecture [5], and HP "The Machine" [2]. Though previous efforts have not fully paned out yet, we believe it is inevitable, as evidenced by the emerging TPU Pods that pack high-density specialized hardware into a rack [8]. Even a traditional rack contains tens of servers and hundreds to thousands of cores, posing a challenge for scheduling microsecond-scale requests.

While dataplane operating systems address intra-server scheduling between multiple *cores*, head-of-line blocking and temporal load imbalance between multiple *servers* arise when the systems scale out. Using a single core for centralized scheduling and queue management is amenable for one server with a few to tens of cores [39]. But the core would quickly become the bottleneck if it were used to queue and schedule requests for a rack with hundreds to thousands of cores.

In this paper, we present RackSched, the first rack-level microsecond-scale scheduler that provides the abstraction of a rack-scale computer (i.e., a huge server with hundreds to thousands of cores) to an external service with network-system co-design. Serving microsecond-scale workloads is particularly challenging because the scheduler needs to *simultaneously* provide high scheduling speed (i.e., high throughput and low latency of the scheduler) and high scheduling quality (i.e., low tail latency to complete requests). Given the scheduling latency of modern OSes on a single server being a few microseconds [19, 39], our goal is to design a rack-level scheduler with comparable scheduling latency that can scale out to hundreds to thousands of cores in a rack. While there has been a long line of work on scheduling and load balancing, existing solutions are not designed for microsecond-scale workloads: software-based solutions [24, 27, 55, 56] suffer from low scheduling throughput and high scheduling latency (at least millisecond-scale); hardware-based ones [25, 51] are coarse-grained (based on five tuple) and server-agnostic, and thus suffer from long tail latency.

The core contribution of RackSched is a novel network-system co-design that *simultaneously* achieves high scheduling speed and high scheduling quality (§2). We propose a two-layer scheduling framework that integrates inter-server scheduling in the top-of-rack (ToR) switch with intra-server scheduling in each server to approximate centralized scheduling policies. This two-layer design, and the line-rate, on-path inter-server scheduling in the ToR switch, are critical for the scheduler to achieve high speed.

To provide high quality scheduling decisions, our key insight is that the two sources of long tail latency—load imbalance and head-of-line blocking—can be decoupled and handled by separate mechanisms. The ToR switch tracks real-time server loads, and schedules requests at *per-request* granularity to realize inter-server load balancing (LB). Each server keeps its own queue, and uses intra-server scheduling to preempt long requests that block pending short ones. It is known that centralized first-come-first-serve (cFCFS) is near-optimal for low-dispersion workloads, and processor sharing (PS) is near-optimal for heavy-tailed workloads or light-tailed workloads with high dispersion [39, 59, 69]. We use a combination of analytical results and simulations to show that our two-layer scheduling framework provides *near-optimal* performance as *centralized* policies, and is robust to different workloads (Figure 1).

Realizing the inter-server scheduler in the ToR switch requires the switch to schedule requests based on *server loads* on *per-request* granularity (§3). Today's stateful network load balancers map connections to servers based the hash of the five tuple [24, 25, 51, 53, 56], which is *static*, and cannot dynamically adapt the server selection under *microsecond-scale* load imbalance. There are three aspects of our approach to address this challenge. (i) We leverage the switch on-chip memory to store server loads, and use the multi-stage processing pipeline to implement power-of-k-choices and to support a variety of practical scheduling requirements, such as multi-queue policies. (ii) We design a *request state table* for *request affinity*, which guarantees the packets of the same request are sent to the same server. To maintain the dynamic mapping between requests and servers, the request state table supports *insert* (after scheduling the first packet), *read* (for sending following packets), and *remove* (when the request is completed) all in the *data plane*. (iii) We leverage *in-network telemetry* (INT) to monitor server loads with minimal overhead. Servers *piggyback* their load information in their *normal* traffic, and the switch tracks the latest reported load for each server.

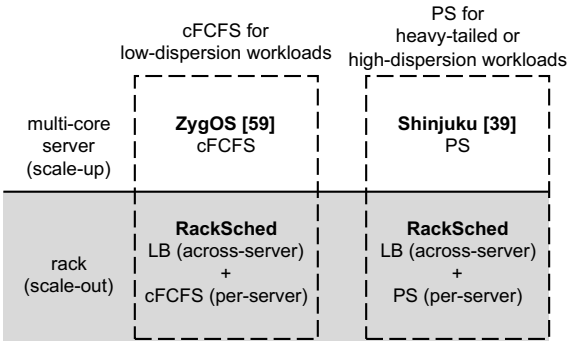


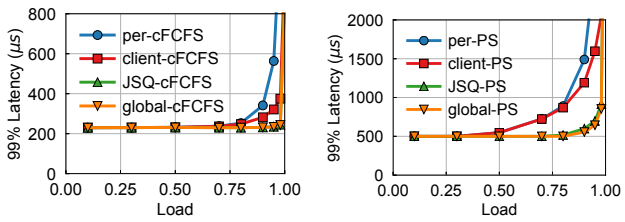
Figure 1: Key idea of RackSched.

Recent switch-based solution R2P2 [42] relies on expensive recirculation which does not scale for high request rate, and its scheduling policy has long tail latency under heavy-tailed or high-dispersion workloads due to head-of-line blocking (§4.5). In addition, R2P2 needs an extra round trip for multi-packet requests to ensure request affinity, while RackSched can finish in one round trip. RackSched is also more general in supporting many practical policies (§3.6).

In summary, we make the following contributions.

- We propose RackSched, the first rack-level microsecond-scale scheduler that provides the abstraction of a rack-scale computer to an external service.
- We design a two-layer scheduling framework that integrates inter-server scheduling in the ToR switch and intra-server scheduling in each server. We use a combination of analytical results and simulations to show that it provides near-optimal performance as centralized policies.
- We design a custom switch data plane for the inter-server scheduler, which realizes power-of-k-choices for near-optimal load balancing, ensures request affinity, and tracks server loads accurately and efficiently.
- We implement a RackSched prototype on a cluster of twelve commodity servers with a Barefoot Tofino switch. End-to-end experiments show that RackSched improves the throughput by up to 1.44×, and scales out the throughput near linearly, while maintaining the same tail latency as one server until the system is saturated.

The code of RackSched is open-source and available at `https://github.com/netx-repo/RackSched`.



(a) Low-dispersion workload. (b) High-dispersion workload.

Figure 2: Simulation results.

2 Design Decisions

In this section, we navigate through the design space of building a microsecond-scale scheduler for rack-scale computers, and highlight the design rationale of RackSched.

Scaling out to a rack. Supporting large application demands requires datacenter systems to scale out to multiple servers in a rack. While existing solutions like Shinjuku [39] solve the problem of scheduling requests to multiple cores (i.e., intra-server scheduling), they do not address the problem of scheduling requests to different servers (i.e., inter-server scheduling). When requests are simply scheduled to the servers randomly, the load imbalance and head-of-line blocking can happen at the server level, causing long tail latency for the entire system.

To motivate our work, we use simulations on representative workloads to show the impact of ineffective scheduling policies. We use the following two request service time distributions: (i) Exp(50) is an exponential distribution with mean = 50 μs, which is representative for low-dispersion workloads; (ii) Trimodal(33.3%-5, 33.3%-50, 33.3%-500) is a trimodal distribution with 33.3% of requests taking 5 μs, 33.3% taking 50 μs and 33.3% taking 500 μs, which is representative for high-dispersion workloads. The simulations assume eight servers and each server has eight workers (cores). The PS time slice used in the simulations is 25 μs.

We first compare the baseline policies that randomly send requests to servers and only use cFCFS or PS inside each server (per-cFCFS and per-PS) with the ideal centralized policies across all workers (global-cFCFS and global-PS). Figure 2 shows that the centralized policies perform better than the baseline policies. The tail latencies of per-cFCFS and per-PS quickly go up when the system load exceeds 0.75 and 0.5 respectively, while global-cFCFS and global-PS keep low tail latencies until the system is almost saturated.

Centralized scheduling cannot scale. The policy comparison in Figure 2 shows that there is a substantial gap between the tail latencies of the centralized policies (global-cFCFS and global-PS) and the baseline policies (per-cFCFS and per-PS). However, directly implementing the centralized policies is challenging because they would require a centralized scheduler for the entire rack. While a single core is capable of running a centralized scheduler to handle the requests for a multi-core server, it is unlikely to scale to a multi-server rack.

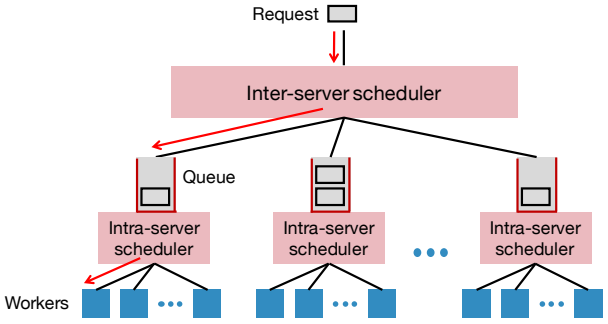


Figure 3: Two-layer scheduling framework.

Indeed, a single scheduler in Shinjuku [39] can scale to up to 11 cores, which falls well short of the demands of a rack with hundreds to thousands of cores.

Hierarchical scheduling. One natural solution to scale up the rack-scale scheduler is a two-layer hierarchical scheduler consisting of an inter-server scheduler at the high level, and per-server schedulers at the low level (Figure 3). This way, the inter-server scheduler only needs to schedule requests across tens of servers, instead of hundreds or thousands of cores. Each server employs its own intra-server scheduler to schedule requests across its cores.

Scaling the inter-server scheduler. While the inter-server scheduler only needs to schedule requests across the servers in the rack (instead of accross all cores), it still needs to handle every request. Assuming a rack with 1000 cores and 10 μs requests, the inter-server scheduler must handle up to 100 millions requests per second (MRPS) to saturate the rack! Unfortunately, such a scheduler would need to process a request every 10 ns, which exceeds the capability of a general-purpose computer.

To address this challenge, in this paper we propose to leverage emerging *programmable switches*, and have the ToR switch implement the inter-server scheduler. This design has the key benefit that the ToR switch is already on the path of the requests sent to the rack, and thus can readily process all these requests at line rate.

JSQ is near optimal and robust. A natural way to approximate a centralized scheduler with a two-layer scheduler is to implement the same scheduler at both layers. For example, if the global scheduler is cFCFS, in the corresponding hierarchical scheduler all the inter-server and intra-server schedulers will be cFCFS as well.

Unfortunately, the cFCFS scheduler needs to maintain a queue, and existing programmable switches have too limited memory to buffer requests, and are not well equipped to maintain dynamic data structures, such as queues. The join-the-shortest-queue (JSQ) scheduler can address the challenge because it is a bufferless scheduler. Upon a request arrival, it immediately forwards the request to the server with the shortest queue. This way, JSQ achieves fine-grained load balancing across the rack's servers. Figure 2 confirms that JSQ-cFCFS

and JSQ-PS can deliver nearly the same performance as the centralized policies (global-cFCFS and global-PS).

Theoretically, the two-layer scheduling framework implements the A/S/K/JSQ/P models in queueing theory, where A is the inter-arrival distribution, S is the service time distribution, K is the number of servers, JSQ is the join-the-shortest-queue policy implemented by the inter-server scheduler, and P is the intra-server scheduling policy which is either cFCFS or PS in this case. In particular, it is known that JSQ provides near-optimal load balancing, and importantly, is *robust* against request service time distributions. An expanded discussion is in the technical report [74].

Approximating JSQ. While conceptually simple, JSQ cannot be implemented in its definite form in practice, because it requires the switch to know the exact queue length of each server when scheduling a request. It takes a round trip time for the switch to ask each server, during which the queue lengths may have changed for microsecond-scale workloads. Furthermore, imperfect JSQ based on delayed server status is prone to *herding*, where several consecutive requests are sent to the same server before the server load is updated, and this can generate micro bursts and degrade system performance. Note that herding here is not caused by multiple asynchronous load balancers as there is only one load balancer (the inter-server scheduler), but from stale server load information in the load balancer. In addition, the switch can only do a limited number of operations for each request, and finding the shortest queue cannot be implemented for many tens of servers. Thus, we use power-of-k-choices to approximate JSQ, which samples k servers for each request and chooses the one with the minimum load. This approximation provides comparable performance as JSQ in theory [18] (see [74]), and handles these practical limitations well.

Why not a distributed, client-based solution? An alternative solution is to implement distributed scheduling at each client. The clients can use JSQ, power-of-k-choices or more complicated solutions like C3 [63] to pick workers for their requests. Such a client-based solution has two drawbacks. First, it needs client modification and increases system complexity. The clients need to probe server loads and make scheduling decisions. More importantly, the clients need to be notified for *every* system reconfiguration (e.g., adding or removing servers), because they have to know which set of servers a request can be sent to. Notifying a large number of clients of the latest system configuration imposes both a consistency challenge and high system overhead. Putting these functionalities in a scheduler, on the other hand, simplifies the clients and avoids these system complexities.

Second, a distributed, client-based solution provides a worse trade-off between performance and probing overhead than a centralized scheduler for microsecond-scale workloads. This is because microsecond-scale workloads are IO-intensive, and a probing request incurs comparable processing cost as a normal request at the servers. Thus, probing needs to be minimized to improve the throughput of processing the actual requests. No matter whether probing is done proactively or piggybacked in reply packets, given n clients with the same sending rate, a centralized scheduler can utilize n times as much probing data as that of one client in a client-based solution, resulting in better scheduling quality. Figure 2 confirms the benefit of the centralized scheduler over a client-based solution with a piggyback-based probing mechanism (client-cFCFS and client-PS). The simulation does not model the client software delay and the network delay to get the server loads, which favors the client-based solution. The client-based solution performs worse in real experiments (§4.5). In a multi-pipeline switch, though states are not shared across pipelines, RackSched can approximate JSQ within each pipeline. It works better than the client-based solution because the number of pipelines (e.g., 4) is far smaller than the number of clients (e.g., 1000 or more).

Putting it all together. We propose a two-layer scheduling framework that integrates inter-server scheduling in the ToR switch and intra-server scheduling in each server. The ToR switch uses power-of-k-choices to achieve inter-server load balancing, and each server uses cFCFS or PS to minimize head-of-line blocking. This solution approximates centralized scheduling for the entire rack, and provides the abstraction of a rack-scale computer: the capacity (throughput) of the rack-scale computer is the sum of that of its servers, and the tail latency is maintained as that of one server.

Challenges. Translating the two-layer scheduling framework to a working system implementation presents several technical challenges:

- What is the system architecture to realize this two-layer scheduling framework?
- How does the switch schedule requests based on the server loads, and handle practical scheduling requirements?
- How does the system ensure request affinity (i.e., the packets of the same request are sent to the same server), when the switch processes each packet independently?
- How do servers expose their states to the switch so that the switch can track the real-time loads on the servers efficiently and accurately?

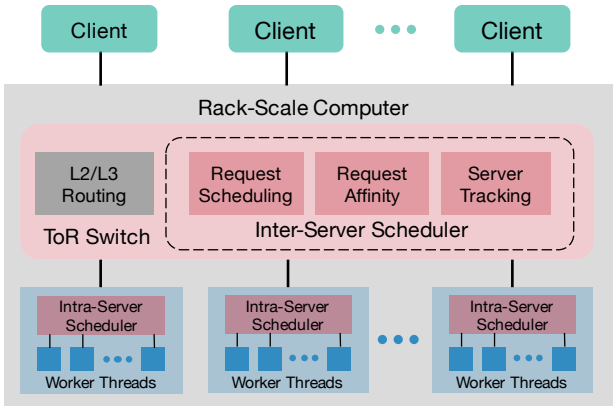
3 RackSched Design

In this section, we present the design of RackSched to address the challenges. We first give an overview of the system architecture, and then describe each component in detail.

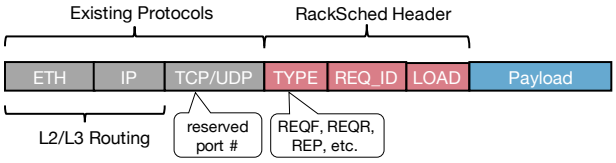
3.1 System Architecture

The core of RackSched is a two-layer scheduling framework that combines inter-server scheduling and intra-server scheduling. Figure 4(a) shows the RackSched architecture.

Inter-server scheduling. The ToR switch performs inter-server scheduling at per-request granularity via three modules:



(a) RackSched architecture.



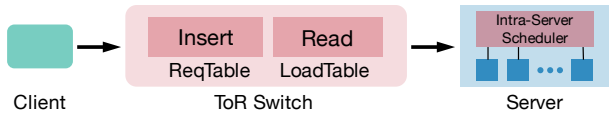
(b) RackSched packet format.

Figure 4: RackSched overview.

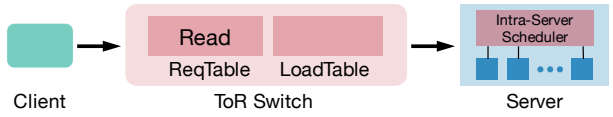
a request scheduling module that selects a server for a new incoming request based on server loads (§3.3) and scheduling requirements (§3.6), a request affinity module that forwards the packets of the same request to the same selected server (§3.4), and a server tracking module that tracks the real-time load on each server (§3.5). All three modules are implemented in the switch data plane that enables the inter-server scheduler to run at line rate.

Intra-server scheduling. Each multi-core server in the rack runs multiple worker threads to process requests. Each server has a centralized scheduler to queue and schedule requests to its own workers. The scheduler implements centralized scheduling policies for intra-server scheduling. Each server also piggybacks its load information in reply messages to report its load to the ToR switch.

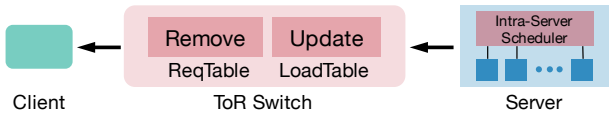
Deployment options. There are two deployment options for RackSched. (*i*) The first one is to integrate RackSched with the ToR switch of a rack-scale computer. This option adds additional functionalities to the ToR switch, but does not change any other part of the datacenter network. (*ii*) The second option is to treat the switch-based scheduler as a specialized server with a programmable switching ASIC. This server can be connected to the same ToR switch as worker servers. It owns the anycast IP address so all requests would be first sent to it for scheduling. By properly updating the addresses, it can also force the reply packets to pass through it before returning to the clients, in order to clear request states and update server loads. This option does not even modify the ToR switch, but has the latency cost of an extra hop by the detour to the switch-based scheduler.



(a) The first packet of a request.



(b) The following packets of the request.



(c) The reply packets.

Figure 5: Request processing in RackSched.

3.2 Request Processing

Network protocol. RackSched is designed for intra-datacenter scenarios. It uses an application-layer protocol which is embedded inside the L4 payload. We reserve an L4 port to distinguish RackSched packets from other packets. The network uses existing L2/L3 routing protocols to forward packets. There are no modifications to the switches in the network other than the ToR switch. The ToR switch uses the reserved L4 port to invoke the custom modules to process RackSched packets. Other switches do not need to understand the RackSched protocol, nor do they need to process RackSched packets. RackSched is orthogonal to and compatible with other network functionalities, such as flow/congestion control, which is typically implemented by the transport layer or the RPC layer (e.g., eRPC [41]). RackSched ensures request affinity under packet loss and retransmission by maintaining connection state (§3.4).

RackSched only requires the applications to add the RackSched header between the TCP/UDP header and the payload (Figure 4(b)), so that it can make scheduling decisions based on the RackSched header. Note that for TCP handshake packets that do not have any payload, the RackSched header should be appended after the TCP header to expose necessary information to the switch. We emphasize that RackSched focuses on microsecond-scale workloads. It is not intended to support long-lived TCP connections, which impose unnecessary system overhead to maintain connection states, especially under switch failures (§3.4), and restrict the scheduler from making per-request scheduling decisions to address temporal load imbalance. RackSched does support request dependency for tasks with multiple requests (§3.6).

The RackSched header contains three major fields, which are `TYPE`, `REQ_ID`, and `LOAD`. `TYPE` indicates the type of the packet, e.g., `REQF` (the first packet of a request), `REQR` (a remaining packet of a request), and `REP` (a reply packet).

Algorithm 1 ProcessPacket(pkt)

– *ReqTable*: on-chip memory for request-server mapping
– *LoadTable*: on-chip memory for server loads

// first packet of a request
1: **if** $pkt.type == REQF$ **then**
2: $server_ip \leftarrow LoadTable.select_server()$
3: $ReqTable.insert(pkt.req_id, server_ip)$
// remaining packets of a request
4: **else if** $pkt.type == REQR$ **then**
5: $server_ip \leftarrow ReqTable.read(pkt.req_id)$
// reply packets
6: **else if** $pkt.type == REP$ **then**
7: $ReqTable.remove(pkt.req_id)$
8: $LoadTable.update(pkt.srcip, pkt.load)$
9: Update packet header and forward

REQ_ID is a unique ID for each request. All packets of the same request and the corresponding reply use the same REQ_ID. To ensure a REQ_ID is globally unique, each client appends its client ID in front of its locally generated unique request ID, i.e., a tuple of <client ID, local request ID>. LOAD indicates the load of the server. The server piggybacks its current queue length in the LOAD field in reply packets. LOAD is not used in request packets.

Processing request packets. Clients use an *anycast* IP address as the destination IP to send requests to the rack-scale computer, and are unaware of the number of servers behind the ToR switch. Figure 5 illustrates how RackSched processes packets, and Algorithm 1 shows the high-level pseudo code of the switch. The switch keeps two essential sets of state in the switch on-chip memory. One is *ReqTable* which stores the mapping from the request IDs to the servers, and the other is *LoadTable* which stores the queue lengths of the servers. As shown in Figure 5(a), when the first packet of a request arrives at the switch, the switch selects a server based on *LoadTable*, and remembers this selection by inserting an entry to *ReqTable* (line 1-3). Then in Figure 5(b), when remaining packets of the request arrive, the switch checks the *ReqTable* to get the selected server (line 4-5), which ensures request affinity. The switch uses the selected server IP to update the destination IP in the packet header and sends the packet to the corresponding server (line 9).

Processing reply packets. After a server receives a request, it uses its local scheduler to schedule and processes the request. Then the server generates a reply, and sets the LOAD field with its current queue length. As shown in Figure 5(c), the switch deletes the mapping from *ReqTable*, because the request has completed and the memory space can be freed for other requests (line 7). The switch also updates the server load in *LoadTable* based on the LOAD field (line 8). We do not distinguish the first and following packets for a reply even if the reply contains *multiple* packets (also equivalent to *multiple* replies). Because *ReqTable* checks *req_id* for deletion, if a slot is reused by another request, the following reply packets of the previous request would not be applied. The updates of *LoadTable* only affect server selection of new requests. Note that this is compatible with TCP even if the client initiates the termination of the connection, as the mapping of this request is removed from *ReqTable* when the server receives the request and sends the first reply packet back to the client. The switch control plane periodically checks the data plane to remove *stale* mappings from *ReqTable*, which can be caused by server failures or lost reply packets. In the end, the switch updates the source IP to the anycast IP in the packet header, and sends the packet back to the client (line 9).

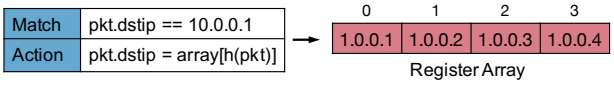


Figure 6: Traditional hash-based random dispatching.

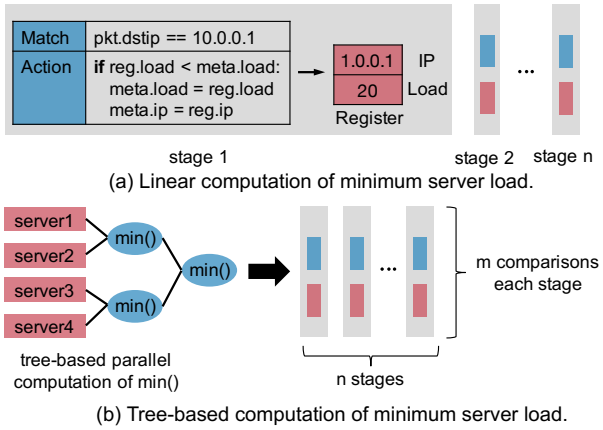


Figure 7: Optimal server selection.

3.3 Request Scheduling

The request scheduling module dynamically schedules requests based on server loads. Unfortunately, this is not supported in today's switches. Today's switch-based load balancers such as SilkRoad [51] only support ECMP-like random dispatching based on the five tuple. Figure 6 shows how hash-based random selection is implemented in the data plane. The register array stores a set of server IPs for the anycast IP 10.0.0.1. The rule in the match-action table matches packets with their destination IP being the anycast IP 10.0.0.1, and the action rewrites the destination IP to an IP in the register array, which is selected by computing a hash on the packet header (usually the five tuple). Because the selection is static, and is purely based on the hash, it can cause load imbalance and long tail latency as discussed in §2. We now describe how to realize dynamic request scheduling based on server loads. Handling practical scheduling requirements is in §3.6.

Optimal server selection. We leverage the register arrays to store the server loads together with the server IPs and use the multi-stage packet processing pipeline to compute the minimum. A naive way is to use multiple stages to scan the

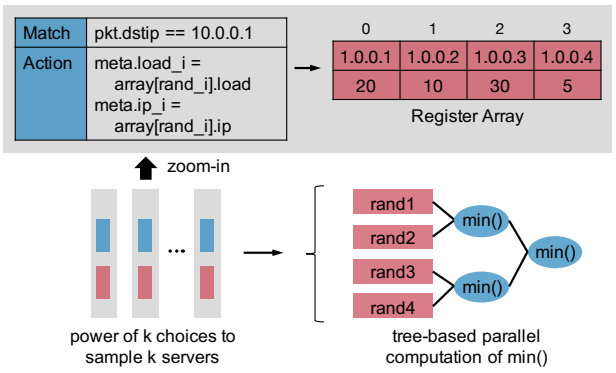


Figure 8: Approximate server selection.

server loads linearly, as shown in Figure 7(a). The number of servers this solution can support is limited to the number of stages. As a switch typically has 10–20 stages and many stages need to be used by other switch functionalities, this solution is not scalable. It can be optimized by computing the minimum in a tree structure as shown in Figure 7(b). The comparisons between two servers in each layer of the tree have no dependencies, and thus can be done in parallel in the same stage. In the ideal case, given n servers, this tree-based solution requires $log(n)$ stages, while the naive solution requires n stages. Let each stage support up to m comparisons. The comparisons in the first few layers need to be distributed to multiple stages, if they are larger than m.

Approximate server selection. As discussed in §2, always choosing the server with the shortest queue is prone to herding, and due to the limited stages and the need to support other functionalities, the tree-based approach cannot scale to many tens of servers. We design an approximate server selection mechanism based on power-of-k-choices [18], i.e., the switch samples k servers and chooses the one with the shortest queue from them. As shown in Figure 8, the sampling can be done via multiple stages if k is bigger than the number of register read operations supported by one stage. After the k servers are sampled, the tree-based mechanism can be applied to get the one with the shortest queue.

3.4 Request Affinity

Request affinity ensures all packets of the same request are sent to the same server. This is challenging because the switch processes each packet independently. In traditional network load balancers [24, 25, 51, 53, 56], the server selection is solely based on the hash of the packet header, and the switch does not need to keep any state for request affinity. But in RackSched, the selection is dynamic. If the switch performs a server selection for every packet, the packets of the same request might be sent to different servers.

Realizing request affinity requires the switch to keep states. Abstractly, the switch should maintain a request state table to store the mapping from request IDs to server IPs (i.e., *ReqTable* in Algorithm 1). One option is to use a match-action table, where the request IDs are stored in the match, and the servers IPs are stored in the action to update the destination IP of the packets (e.g., used by SilkRoad [51]). This option, however, does not work for microsecond-scale requests at million RPS throughput, because updating the match-action table (e.g., adding or removing a request) requires the control plane, which can only do about 10K updates per second [36, 48, 52]. To address this challenge, our design leverages register arrays to realize a multi-stage hash table that implements all necessary operations (i.e., *insert*, *read* and *remove*) for *ReqTable* in the data plane, as shown in Figure 9. Unlike match-action tables, register arrays can only be accessed via an index. We use the hash of the request ID to find the slot for a request, and the slot stores the request state, i.e., the request ID and server IP. To handle hash collisions and the limited array size in each stage, we leverage multiple stages to build a multi-stage hash table. Algorithm 2 shows the pseudo code to implement the three operations on *ReqTable* in Algorithm 1. The switch iterates over the stages to find an empty slot to insert a new request (line 1-5), and to find a matched slot to read the server IP (line 6-9) or remove a completed request (line 10-14). RackSched does not decrease the capability of the system to defend against DoS attacks. The switch has sufficient memory for *ReqTable* to support high throughput (§4.1), and a DoS attack that overwhelms *ReqTable* could have overwhelmed the servers first.

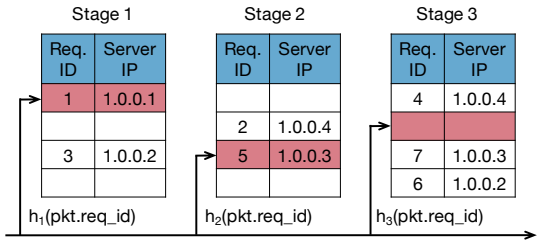


Figure 9: Multi-stage hash table for request affinity.

Handling switch failure. There is a relevant notion to request affinity called Per-Connection Consistency (PCC) for stateful layer-4 load balancers, which requires a TCP connection to be kept across load balancer failures and system reconfigurations [24, 25, 51, 53, 56]. We emphasize that RackSched focuses on microsecond-scale requests with strict deadlines (e.g., a couple of the request execution time). Rebooting a failed switch or replacing it with a backup switch takes a few minutes, by the time of which the requests have already *missed* their deadlines. Therefore, different from PCC, maintaining request affinity across switch failures is a non-goal for RackSched. Because of the *fate sharing* between the ToR switch and the rack, it is safe to disregard the *ReqTable* upon a switch failure, and the new switch starts with an empty *ReqTable*. Note that RackSched does not increase the chance of switch failures as a normal ToR switch without RackSched can still fail and make the rack disconnected.

Algorithm 2 Request affinity

– *ReqTable*[*n*][*m*]: register arrays to store request state, which spans *n* stages and has *m* slots in each stage

```
1: function INSERT(req_id, server_ip)
2:     for stage i in all stages do
3:         if ReqTable[i][h(req_id)] == None then
4:             ReqTable[i][h(req_id)] ← (req_id, server_ip)
5:             return
6: function READ(req_id)
7:     for stage i in all stages do
8:         if ReqTable[i][h(req_id)].req_id == req_id then
9:             return ReqTable[i][h(req_id)].server_ip
10: function REMOVE(req_id)
11:     for stage i in all stages do
12:         if ReqTable[i][h(req_id)].req_id == req_id then
13:             ReqTable[i][h(req_id)] ← None
14:     return
```

Handling system reconfiguration. Unlike switch failures, there is no fate sharing between each server and the rack, and RackSched does maintain request affinity across system reconfigurations such as adding or removing servers for an application. Because RackSched uses *ReqTable* to store the mapping, ongoing requests simply check *ReqTable* to go to the correct servers. Only the request scheduling module (§ 3.3) needs to be updated to have the right set of servers to choose from for new requests. We pre-allocate a large number of registers for *LoadTable* at compilation time, and use another register to indicate the number of *active* servers, which is dynamically updated for system reconfigurations. For an unplanned server removal (e.g., a server failure), RackSched uses the switch control plane to update the *ReqTable* and delete the stale entries related to the removed server.

3.5 Server Tracking

The server tracking module updates the server loads (i.e., *LoadTable* in Algorithm 1) for the switch to make scheduling decisions. The challenge is to accurately track the server loads at real-time with low overhead. A straightforward solution is to let the switch control plane periodically poll the queue lengths at each server and update the data plane. However, due to the millisecond-scale delay and the limited rate of control plane updates [37, 52], this solution does not apply to the microsecond-scale workloads targeted by RackSched. To do this in the data plane, a possible solution is to let the switch *proactively* track the server loads, i.e., incrementing and decrementing the counters for queue lengths when processing request and reply packets. This solution suffers from estimation errors due to packet loss and retransmissions, and fixes like decreasing the counters based on the server processing rate [47] cannot handle temporal load imbalance in high-dispersion microsecond-scale workloads.

RackSched leverages in-network telemetry to accurately track server loads with minimal overhead. In-network telemetry is widely used in network monitoring and diagnosis where switches put relevant measurement data into packet headers in the data plane. RackSched applies this mechanism to track server loads. Then the servers piggyback their loads in the reply packets to update the counters in the switch, which does not introduce new packets and thus minimizes system overhead. A potential problem is the feedback loop delay as stale information can cause herding, which can degrade the scheduling performance and make the system unstable (i.e., swing between overloading different servers). In RackSched, the switch and the servers are directly connected in the same rack, and the server-side data plane implementation bypasses traditional TCP/IP stack to report its queue length to the switch quickly, making the feedback loop delay minimal. And together with power-of-k-choices scheduling, RackSched can effectively avoid herding. An alternative solution that only keeps the server with the minimum load in the switch and updates it based on in-network telemetry cannot leverage power-of-k-choices to avoid herding. We show the impact of different ways to track and represent server loads in §4.6.

3.6 Handling Scheduling Requirements

Multi-queue support. By default, RackSched uses a single-queue policy, i.e., the system does not differentiate a priori between request types and aims to meet a single SLO for tail latency (e.g., a photo caching workload with only *get* requests). RackSched also supports multiple queues if the workload has multiple request types that have distinct service time distributions (e.g., a key-value store workload with both *get* and *range* requests). Applications indicate the request type in the `TYPE` field of the packet header. Each server maintains a separate queue for each type for intra-server scheduling. The switch maintains the counters for each type in *LoadTable*, and schedules requests based on the queue lengths of the request type. We remark that there is no fundamental limit on the number of queues on each server, since the queues are implemented in software and the switch only needs to keep a counter for each queue on each server.

Locality and placement constraints. RackSched handles two types of common locality and placement constraints, which are data locality and request dependency. (i) Data locality requires a request to be processed on a subset of servers that hold the input data. To support data locality, applications set different `LOCALITY` values to represent different locality requirements (i.e., different sets of servers that can process this request), and the switch maintains different mappings, which map a server ID to a server for different `LOCALITY` values. (ii) Request dependency requires multiple requests to be scheduled to the same server, e.g., a task consists of multiple requests and the input of one request is the output of one or more other tasks. Request dependency is supported using request affinity: relevant requests carry the same `REQ_ID` in their headers, so that they will be sent to the same server. Additional information is included in the RackSched header for the number of subsequent requests to expect. The server can send replies to each request separately and independently.

RackSched only requires the server to set `TYPE` to be reply in replies after it has received all requests in the set in order to safely delete the state in the switch. Note that applications can still use different request IDs for different requests and receive replies as soon as some requests are completed, by adding application-specific metadata in the payload.

Resource allocation policies. The scheduler is responsible for allocating resources when the demand exceeds the capacity of the rack-scale computer. RackSched supports two types of common resource allocation policies, which are strict priority and weighted fair sharing. (i) To support strict priority, each server maintains a separate queue for each priority. Similar to the multi-queue support, the switch tracks the queue lengths, and balances the server loads for each priority. Each server uses intra-server scheduling to preempt low-priority requests when high-priority requests arrive, which can be done in 5 μs in our implementation based on Shinjuku [39]. (ii) Supporting weighted fair sharing is similar. Each server maintains a separate queue for each client, and performs weighted fair queueing [23] for intra-server scheduling on the granularity of slice in PS. The switch tracks the queue lengths and balances the server loads for each client.

4 Evaluation

In this section, we evaluate RackSched with a variety of synthetic and real application workloads. We provide additional experiment results, including locality constraints, priority policies and multiple applications, in the technical report [74].

4.1 Methodology

Testbed. The experiments are conducted on a testbed of twelve server machines connected by a 6.5Tbps Barefoot Tofino switch. Each server has an 8-core CPU (Intel Xeon E5-2620 @ 2.1GHz), 64GB memory, and one 40G NIC (Intel XL710). Eight servers are used as workers to process requests, and they run Shinjuku [39] with our extension. Four servers are used as clients to generate requests. The bottleneck of the system is at the workers.

Implementation. We have implemented a RackSched prototype and integrated it with Shinjuku [39]. (i) The switch data plane is written in P4 [16] and compiled to Barefoot Tofino ASIC [11] with P4 Studio [10]. The request state table contains a hash table with 64K slots. The default implementation uses power-of-2-choices. (ii) The worker server is based on Shinjuku [39]. We have extended Shinjuku to support the RackSched packet header, maintain a counter and update the counter upon request arrival and reply departure to track the queue length and append the queue length in reply packets. Both RackSched and Shinjuku preempt requests that exceed 250 μs in our experiments. (iii) The client is open-loop and implemented in C using Intel DPDK 16.11.1 [4]. It can generate requests at high request rate based on synthetic workloads and the RocksDB application, and measure the throughput and latency of RackSched.

Resource consumption. Our prototype uses 13.12% SRAM, 9.96% Match Input Crossbar, 12.5% Hash Unit and 25% Stateful ALUs of the Tofino ASIC resources. We provide an analysis and a back-of-the-envelop calculation to show that RackSched consumes little switch memory. RackSched has two sets of state on the switch, i.e., *LoadTable* and *ReqTable*. (i) *LoadTable* maintains a counter for each queue of each server. Let the counter be 4 bytes, the number of queues in each server be 3 and the number of servers be 32. It only consumes 384 bytes. (ii) *ReqTable* maintains the selected server IPs for the *ongoing* requests, not *all* requests the system have received and processed. Each slot can be *reused* by many times each second because the requests are microsecond-scale. Given an average request processing latency of 50 μs, a slot can support 20 KRPS throughput, and a table with 64K slots can support 1.28 BRPS throughput. Let the request ID and server IP both be 4 bytes. A table with 64K slots (our implementation) consumes 256 KB, which is only a few percent of the on-chip memory (tens of MB). Overflowed requests can fall back to hash-based random dispatching which preserves request affinity.

Workloads. We use a combination of synthetic and application workloads. They include the following workloads. By default, the workloads use one-packet requests.

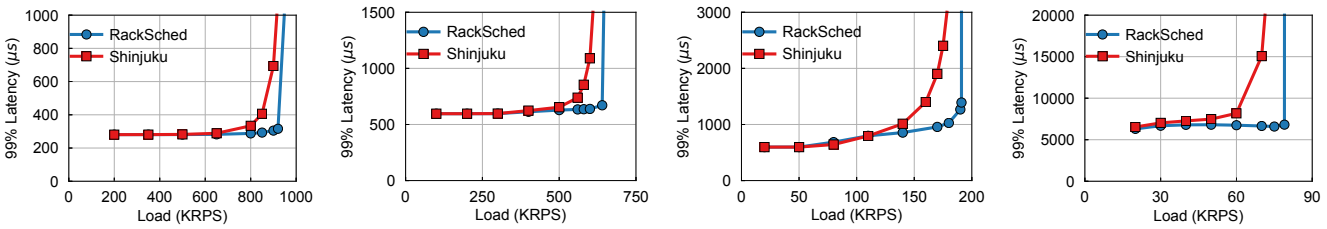
- Exp(50) is an exponential distribution with mean = 50 μs, which represents common query and storage workloads, such as *get* requests in photo caching.
- Bimodal(90%-50, 10%-500) is a bimodal distribution with 90% of requests taking 50 μs and 10% taking 500 μs, which represents workloads with a mix of simple requests and complex requests, such as *get* and *range* requests in key-value stores.
- Bimodal(50%-50, 50%-500) is a bimodal distribution with 50% of requests taking 50 μs and 50% taking 500 μs, which represents workloads with half simple requests and half complex requests.
- Trimodal(33.3%-50, 33.3%-500, 33.3%-5000) is a trimodal distribution with a third of requests taking 50 μs, 500 μs and 5000 μs, respectively, which represents workloads with more diverse request types, such as *point*, *range* and complex *join* requests in databases.

We also use RocksDB 5.13 [60], an open-source production-quality key-value store, as a real application workload to evaluate RackSched. RocksDB is configured to store data in DRAM to avoid blocking behavior and achieve low latency.

4.2 Synthetic Workloads

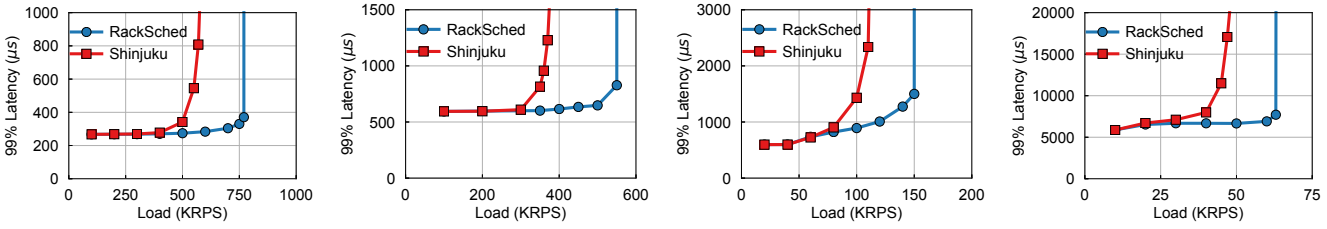
We evaluate the system on synthetic workloads that cover large application space. We compare RackSched with that directly runs Shinjuku in the cluster, i.e., the requests are randomly sent to the servers.

Figure 10(a) and Figure 10(b) compare RackSched and Shinjuku under Exp(50) and Bimodal(90%-50, 10%-500) workloads, respectively. In these two figures, both RackSched



(a) Exp(50). (b) Bimodal(90%-50, 10%-500). (c) Bimodal(50%-50, 50%-500). (d) Trimodal(33.3%-50, 33.3%-500, 33.3%-5000).

Figure 10: Experimental results for synthetic workloads with homogeneous servers.



(a) Exp(50). (b) Bimodal(90%-50, 10%-500). (c) Bimodal(50%-50, 50%-500). (d) Trimodal(33.3%-50, 33.3%-500, 33.3%-5000).

Figure 11: Experimental results for synthetic workloads with heterogeneous servers.

and Shinjuku use a single-queue policy. Under Exp(50), the 99% latencies of RackSched and Shinjuku are similar at low load. But the 99% latency of Shinjuku quickly goes up after 800 KRPS, while that of RackSched can support the system load up to 950 KRPS. Under Bimodal(90%-50, 10%-500), the 99% latency of Shinjuku quickly increases after 500 KRPS, while that of RackSched stays stable until 650 KRPS. In both workloads, RackSched supports larger request load with lower tail latency, because its inter-server scheduling addresses temporal load imbalance between servers, while Shinjuku experiences short bursts and long queues in individual servers under high request load.

Figure 10(c) and Figure 10(d) show the results for Bimodal(50%-50, 50%-500) and Trimodal(33.3%-50, 33.3%-500, 33.3%-5000) workloads, respectively. In these two figures, both RackSched and Shinjuku have a separate queue for each request type. Again, RackSched significantly outperforms Shinjuku. The improvement of RackSched is larger in Trimodal(33.3%-50, 33.3%-500, 33.3%-5000) than Bimodal(50%-50, 50%-500), because Trimodal(33.3%-50, 33.3%-500, 33.3%-5000) has more diverse service times and can benefit more from effective inter-server scheduling.

Figure 11 shows the results with heterogeneous servers. In this case, four servers have four workers and the other four servers have seven workers (one core used by the scheduler). This evaluates the cases when some servers are slower or some cores of these servers are grabbed for other purposes [15, 54]. We compare RackSched with Shinjuku under the same four distributions in Figure 10. RackSched is load-aware and tends to send requests to the servers with shorter queue lengths, while Shinjuku distributes the requests to the servers uniformly, disregarding the heterogeneity. RackSched can improve the performance further with heterogeneous servers.

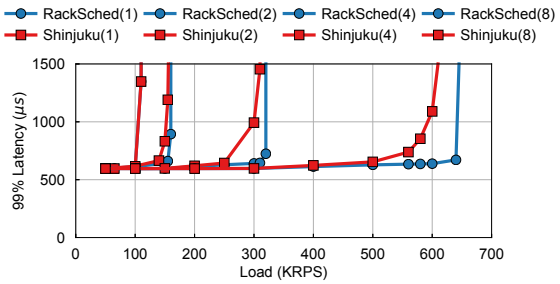
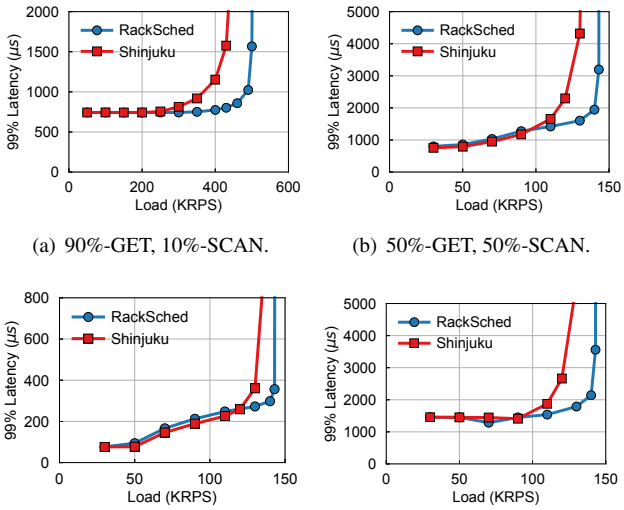


Figure 12: Scalability results.

4.3 Scalability

The key benefit of RackSched is that it enables the system to scale out by adding servers, while achieving low tail latency at high throughput. Figure 12 shows the 99% latency under different request load with one, two, four and eight servers, respectively. This figure uses Bimodal(90%-50, 10%-500) workload, and the results for other workloads are similar. With one server, the two systems, i.e., RackSched (1) and Shinjuku(1), have the same performance, as there is no need for inter-server scheduling. With two servers, load imbalance can happen, but the variability is small. With four servers, micro bursts can cause bigger temporal load imbalance, and the improvement of inter-server scheduling is also bigger. When there are eight servers, there is more variability between the loads on the servers, and inter-server scheduling has more opportunities to improve performance. Shinjuku(8) can only maintain low tail latency until 500 KRPS, while RackSched (8) can maintain low tail latency until 650 KRPS. We expect the improvement of RackSched over Shinjuku would be larger with more servers, because there would be more variabilities with more servers.



(a) 90%-GET, 10%-SCAN. (b) 50%-GET, 50%-SCAN.

(c) GET in 50%-GET, 50%-SCAN. (d) SCAN in 50%-GET, 50%-SCAN.

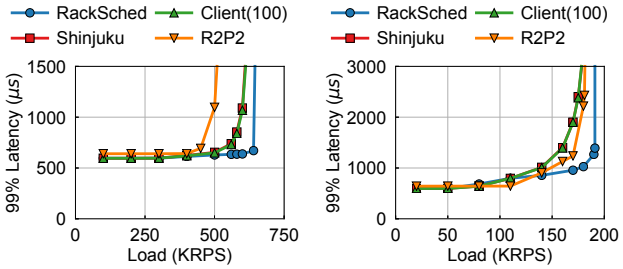
Figure 13: Experimental results for RocksDB.

Overall, RackSched scales out the total throughput of the system near linearly with the number of servers in the rack. And the throughput improvement is achieved without increasing the tail latency. Even with more servers, RackSched is still able to maintain the same tail latency as one server until the system is saturated.

4.4 Application: RocksDB

We use RocksDB [60] to demonstrate the benefits of RackSched on real applications. RocksDB is an open-source production-quality storage system that is widely deployed to support many online services such as Facebook. In the experiments, RocksDB is configured with an in-memory file system (/tmpfs/) for microsecond-scale request processing. We use two request types. One is *GET* which gets 60 objects with a median request service time of 50 μs. The other is *SCAN* which scans 5000 objects with a median service processing time of 740 μs. Only 326 lines of code are needed to port RocksDB to RackSched and Shinjuku. Figure 13(a) shows the results for the workload that contains 90% *GET* requests and 10% *SCAN* requests. In this experiment, the system uses a single-queue policy. At low request load, RackSched and Shinjuku have comparable 99% latency. But Shinjuku can only maintain low tail latency until 300 KRPS, while RackSched is able to keep low tail latency until 500 KRPS.

Figure 13(b) shows the results for the workload that contains 50% *GET* requests and 50% *SCAN* requests. In this experiment, the system uses a multi-queue policy. RackSched is able to maintain low tail latency at a higher request load than Shinjuku. We further break down the results for each request type for this workload. Figure 13(c) and Figure 13(d) show the 99% latency for *GET* and *SCAN* under different total request load, respectively. Because RackSched uses the switch to balance the load of each request type between the



(a) Bimodal(90%-50,10%-500). (b) Bimodal(50%-50,50%-500).

Figure 14: Comparison with other solutions.

servers, the improvement of RackSched over all requests does not come at the cost of sacrificing any individual request type. For both request types, RackSched is able to deliver comparable tail latency at low load, achieve significantly lower tail latency at high load, and support higher total request load.

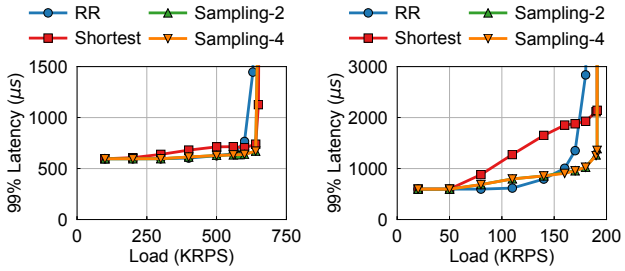
4.5 Comparison with Other Solutions

R2P2 [42] is a recent solution that proposes a join-bounded-shortest-queue (JBSQ) policy for request scheduling, and the solution can be implemented on programmable switches. R2P2 does not have preemptive intra-server scheduling and has head-of-line blocking. Thus, it suffers from long tail latency, especially under high-dispersion workloads. Client-based solutions are lack of global view and use power-of-k-choices scheduling based on stale server load information, and thus they suffer from inaccurate scheduling decisions. We emulate 100 clients that generate requests with the same rate in the machines. Each client performs the same policy as RackSched and tracks server queue lengths via piggybacking by its own. The performance of Client(10) (which emulates 10 clients) and Client(1000) (which emulates 1000 clients) are nearly the same as that of Client(100). Figure 14 shows the performance of RackSched, Shinjuku, the client-based solution and R2P2 under Bimodal(90%-50, 10%-500) and Bimodal(50%-50, 50%-500) workloads. In both workloads, RackSched outperforms others by maintaining low latency at higher request rate, and Client(100) has nearly the same performance as Shinjuku. More importantly, R2P2 is not robust to service time distributions. It is close to RackSched under Bimodal(50%-50, 50%-500), and the gap between R2P2 and RackSched is significantly larger under Bimodal(90%-50, 10%-500).

4.6 Analysis of RackSched

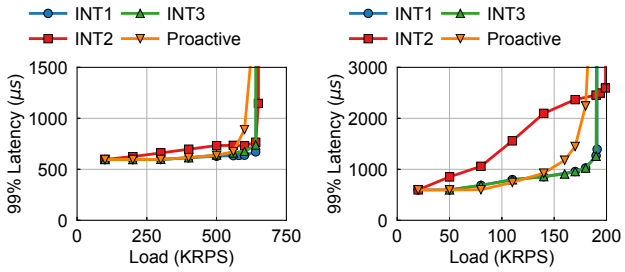
We analyze RackSched and show the impact of different design choices, including different scheduling policies of the switch-based inter-server scheduler and different mechanisms to track the server loads.

Impact of switch scheduling policies. Figure 15 evaluates the impact of different scheduling policies under Bimodal(90%-50, 10%-500) and Bimodal(50%-50, 50%-500) workloads. We compare four scheduling policies: *RR* (which



(a) Bimodal(90%-50,10%-500). (b) Bimodal(50%-50,50%-500).

Figure 15: Impact of switch scheduling policies.

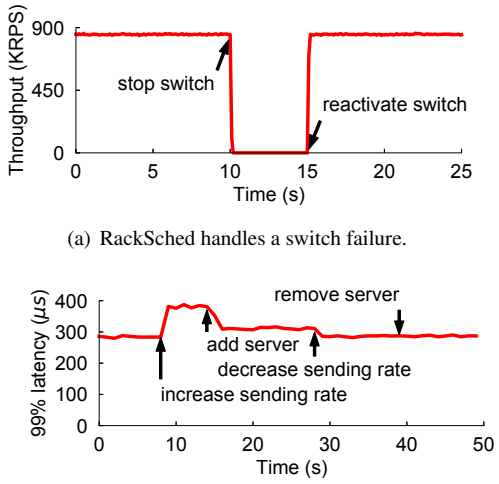


(a) Bimodal(90%-50,10%-500). (b) Bimodal(50%-50,50%-500).

Figure 16: Impact of server load tracking mechanisms.

schedules requests to server with round-robin), *Shortest* (which chooses the server with the smallest queue length), *Sampling*-2 (which samples two servers and chooses the one with the smaller queue length), and *Sampling*-4 (which samples four servers and chooses the one with the smallest queue length). RR sends an even number of requests to each server, without considering the variability of request service times. Thus, it suffers from long tail latency at high request load. Theoretically, *Shortest* can provide effective load balancing, but it incurs high tail latency in practice, even at low request load. As discussed in §2, the reason is that there is a delay to update the queue lengths in the switch from the servers. When a server becomes the one with the smallest queue length, multiple consecutive requests would all choose this server, causing a micro *herding* behavior. And the queue length of this server has to wait to be updated until the new queue length is piggybacked in the first reply packet to update in the switch. As discussed in §2, this herding behavior can be handled by adding randomization to the scheduling process. The results in the figure confirm the effectiveness of sampling. For the scale of the evaluated scenario, sampling two and four servers have similar performance, because sampling two servers already provides enough choices to avoid hotspots and enough randomization to avoid herding.

Impact of server load tracking mechanisms. Figure 16 evaluates the impact of different mechanisms to track server loads, under both Bimodal(90%-50, 10%-500) and Bimodal(50%-50, 50%-500) workloads. We compare three tracking mechanisms discussed in §3.5: *INT*1 (which tracks the number of outstanding requests for each server and computes the minimum), *INT*2 (which only tracks the minimum number of outstanding requests and updates on reply packets), *INT*3 (which tracks the total service time of outstanding requests for each server) and *Proactive* (which increments and decrements the counters by the switch). *Proactive* cannot precisely maintain the queue length for each server as packet loss and retransmissions can introduce errors on the counters, and as a result, it does not work well as others. *INT*2 performs worse than *INT*1 because it only keeps one server with the minimum load, resulting in herding. *INT*3 is comparable to *INT*1. However, it presumes that the service times are known as a priori, which is normally not the case in practice. *INT*1 works the best because it accurately tracks server loads, enables randomization to avoid herding for effective load balancing, and does not require any priori knowledge.



(a) RackSched handles a switch failure.

(b) RackSched handles server reconfigurations.

Figure 17: Handling failures and reconfigurations.

4.7 Request Affinity

Handling switch failures. To simulate a switch failure, we first stop the switch manually, then reactivate the switch after several seconds. Figure 17(a) shows the total throughput during this period under Exp(50) workload. At 10 s, the switch is stopped and the total throughput drops to 0. We reactivate the switch after 5 seconds and the total throughput recovers to the initial level. The microsecond-scale requests have already timed out after 5 seconds. So it is safe to start with an empty *ReqTable* after the reactivation, as discussed in §3.4.

Handling system reconfigurations. RackSched maintains request affinity during system reconfigurations. Figure 17(b) shows the 99% latency under system reconfigurations. We use two-packet requests under Exp(50) workload, and start with 500 KRPS load and seven machines as the servers. At time 8 s, we increase the request sending rate, and the 99% latency goes up to around 380 μs. At time 14 s, we add another server to serve the requests, and the 99% latency drops to 310 μs. At

time 28 s, we set the request sending rate back to 500 KRPS. And the 99% latency drops to 280 μs further. At time 39 s, we remove a server from the rack. Since seven servers are enough for such workload, the 99% latency remains the same. As discussed in § 3.4, the request affinity is maintained by the *ReqTable* in the above process.

5 Discussion

Target workloads. RackSched supports both stateless and replicated stateful services. Examples include microservices, function-as-a-service, stream processing, replicated caches and storage, and replicated machine learning models for high-throughput inference. It is unlikely for a stateful service to be replicated to all servers in a rack. We expect a more practical scenario is that a rack would run multiple such services, where each service is provided by a subset of (overlapping) servers, i.e., locality constraints. The evaluation shows that RackSched can provide significant improvements for a service hosted on just 8 servers (§4). And we provide additional results to show the benefits for multiple services with locality constraints in the technical report [74]. These results demonstrate that RackSched provides significant benefits even when the service is replicated on just a couple of servers.

Going beyond a rack. We focus on a single rack in this paper. A modern rack can already pack hundreds of cores, and a future rack is expected to pack thousands of cores [1, 2, 5], which is sufficient for many services. For planetary-scale services, a single microservice may span multiple racks. In this scenario, there is no central place like the ToR switch in a rack that can see and process all traffic. Yet, the abstraction of a rack-scale computer provided by RackSched provides a useful building block for distributed inter-rack scheduling. This would be an interesting direction for future work.

6 Related Work

Dataplane designs for low latency. Conventional networking stacks and operating systems usually sacrifice low latency for generality. To address the need for low latency, various dataplane designs have been proposed, including optimized networking stacks [4, 21, 34] and dataplane operating systems [14, 20, 32, 39, 54, 58, 59]. RackSched leverages such dataplane designs and enhances them with an inter-server scheduler, realizing low latency in a rack-scale computer.

Scheduling and resource management. There is a long line of research on job scheduling and resource management [26, 28, 29, 32, 38, 40, 42, 55, 57, 65, 66, 71]. Many systems focus on large jobs that can run from seconds to hours, and they can afford running sophisticated scheduling algorithms to make effective decisions. RackSched works at microsecond scale and optimizes the tail latency with network-system co-design.

Programmable switches. Programmable switches bring new opportunities to improve datacenter networks and systems, such as key-value stores [36, 48, 49, 50], coordination and consensus [35, 45, 46, 70, 73], network telemetry [3, 33], machine learning acceleration [61, 62] and query processing offload [44]. There are also proposals for managing systems built with programmable switches [30, 68, 72]. RackSched is a new solution that leverages the programmable switch as an inter-rack scheduler to optimize microsecond-scale tail latency for rack-scale computers.

7 Conclusion

We present RackSched, a rack-level microsecond-scale scheduler that provides the abstraction of a rack-scale computer to an external service. RackSched leverages a two-layer scheduling framework to achieve scalability and low tail latency. We hope that with the end of Moore's law and Dennard's scaling, RackSched will inspire a new generation of datacenter systems enabled by domain-specific hardware and hardware-software co-design.

Acknowledgments We thank our shepherd Ryan Stutsman and the anonymous reviewers for their valuable feedback. We thank Jack Humphries for helping debug Shinjuku issues. This work is supported in part by NSF grants CNS-1813487, CCF-1918757 and CNS-1955487, a Facebook Communications & Networking Research Award, and a Google Faculty Research Award.

References

[1] FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. https://www.usenix.org/node/179918.

[2] HP The Machine. https://www.labs.hpe.com/the-machine.

[3] In-band Network Telemetry (INT) Dataplane Specification. https://p4.org/specs/.

[4] Intel Data Plane Development Kit (DPDK). http://dpdk.org/.

[5] Intel Rack Scale Design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[6] Memcached key-value store. https://memcached.org/.

[7] Redis data structure store. https://redis.io/.

[8] TPU Pods. https://cloud.google.com/tpu/.

[9] Voltdb in-memory database. https://www.voltdb.com.

[10] Barefoot P4 Studio. https://www.barefootnetworks.com/products/brief-p4-studio/.

[11] Barefoot Tofino. https://www.barefootnetworks.com/technology/.

[12] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 2017.

[13] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 2003.

[14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX OSDI*, 2014.

[15] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *USENIX OSDI*, 2018.

[16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, July 2014.

[17] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Putting the "micro" back in microservice. In *USENIX ATC*, 2018.

[18] M. Bramson, Y. Lu, and B. Prabhakar. Randomized load balancing with general service time distributions. *ACM SIGMETRICS performance evaluation review*, 38(1):275–286, 2010.

[19] F. Cerqueira and B. Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2013.

[20] A. Daglis, M. Sutherland, and B. Falsafi. RPCValet: NI-driven tail-aware balancing of μs-scale RPCs. In *ACM ASPLOS*, 2019.

[21] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX NSDI*, 2018.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, February 2013.

[23] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM*, 1989.

[24] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX NSDI*, 2016.

[25] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *ACM SIGCOMM*, August 2015.

[26] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI*, 2011.

[27] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *USENIX OSDI*, 2016.

[28] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *USENIX OSDI*, 2016.

[29] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *USENIX OSDI*, 2016.

[30] D. Hancock and J. Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *ACM CoNEXT*, 2016.

[31] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 2019.

[32] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *ACM SIGCOMM HotNets Workshop*, 2019.

[33] N. Ivkin, Z. Yu, V. Braverman, and X. Jin. QPipe: Quantiles sketch fully in the data plane. In *ACM CoNEXT*, December 2019.

[34] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *USENIX NSDI*, 2014.

[35] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX NSDI*, 2018.

[36] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, 2017.

[37] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, 2014.

[38] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *USENIX OSDI*, 2016.

[39] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX NSDI*, 2019.

[40] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis. Centralized core-granular scheduling for serverless functions. In *ACM Symposium on Cloud Computing*, 2019.

[41] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *USENIX NSDI*, 2019.

[42] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *USENIX ATC*, 2019.

[43] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *USENIX OSDI*, 2018.

[44] A. Lerner, R. Hussein, P. Cudre-Mauroux, and U. eXascale Infolab. The case for network accelerated query processing. In *CIDR*, 2019.

[45] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM SOSP*, October 2017.

[46] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*, November 2016.

[47] J. Li, J. Nelson, X. Jin, and D. R. Ports. Pegasus: Load-aware selective replication with an in-network coherence directory. *Technical Report UW-CSE-18-12-01*, 2018.

[48] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *USENIX NSDI*, March 2016.

[49] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *ACM ASPLOS*, April 2017.

[50] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST*, 2019.

[51] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[52] NoviSwitch. `http://noviflow.com/products/noviswitch/`.

[53] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with Beamer. In *USENIX NSDI*, 2018.

[54] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX NSDI*, 2019.

[55] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *ACM SOSP*, 2013.

[56] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. In *SIGCOMM CCR*, 2013.

[57] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[58] S. Peter, J. Li, I. Zhang, D. R. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *USENIX OSDI*, 2013.

[59] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *ACM SOSP*, 2017.

[60] RocksDB. RocksDB. `https://rocksdb.org/`.

[61] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *ACM SIGCOMM HotNets Workshop*, November 2017.

[62] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv*, 2019.

[63] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX NSDI*, 2015.

[64] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *ACM SOSP*, 2013.

[65] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.

[66] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *ACM Symposium on Cloud Computing*, 2013.

[67] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How Facebook serves the social graph. In *ACM SIGMOD*, May 2012.

[68] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. Ports, and A. Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX HotCloud Workshop*, July 2020.

[69] A. Wierman and B. Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[70] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin. Netlock: Fast, centralized lock management using programmable switches. In *ACM SIGCOMM*, 2020.

[71] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX OSDI*, 2008.

[72] P. Zheng, T. Benson, and C. Hu. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.

[73] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment*, 2019.

[74] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A microsecond-scale scheduler for rack-scale computers (technical report). In *arXiv*, 2020.

Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale

Shaohong Li, *Google LLC*
Xi Wang, *Google LLC*
Xiao Zhang, *Google LLC*
Vasileios Kontorinis, *Google LLC*
Sreekumar Kodakara, *Google LLC*
David Lo, *Google LLC*
Parthasarathy Ranganathan, *Google LLC*

Abstract

As the demand for data center capacity continues to grow, hyperscale providers have used power oversubscription to increase efficiency and reduce costs. Power oversubscription requires power capping systems to smooth out the spikes that risk overloading power equipment by throttling workloads. Modern compute clusters run latency-sensitive serving and throughput-oriented batch workloads on the same servers, provisioning resources to ensure low latency for the former while using the latter to achieve high server utilization. When power capping occurs, it is desirable to maintain low latency for serving tasks and throttle the throughput of batch tasks. To achieve this, we seek a system that can gracefully throttle batch workloads and has task-level quality-of-service (QoS) differentiation.

In this paper we present Thunderbolt, a hardware-agnostic power capping system that ensures safe power oversubscription while minimizing impact on both long-running throughput-oriented tasks and latency-sensitive tasks. It uses a two-threshold, randomized unthrottling/multiplicative decrease control policy to ensure power safety with minimized performance degradation. It leverages the Linux kernel's CPU bandwidth control feature to achieve task-level QoS-aware throttling. It is robust even in the face of power telemetry unavailability. Evaluation results at the node and cluster levels demonstrate the system's responsiveness, effectiveness for reducing power, capability of QoS differentiation, and minimal impact on latency and task health. We have deployed this system at scale, in multiple production clusters. As a result, we enabled power oversubscription gains of 9%–25%, where none was previously possible.

1 Introduction

Data centers form the backbone of popular online services such as search, streaming video, email, social networking, online shopping, and cloud. The growing demand for online services forces hyperscale providers to commit massive capital to continuously expand their data center fleet. The overall capital expenditures for just the top 5 hyperscale providers (Amazon, Google, Microsoft, Facebook, Apple) in 2019 reached $120B out of a total $210B for all data centers worldwide [10, 11]. The majority of these investments are allocated towards buying and building infrastructure, such as buildings, power delivery, and cooling, to host the servers that compose the warehouse-scale computer. Power oversubscription is the practice of deploying more servers in a data center than the data center's power supply can nominally support if all servers were 100% utilized. Power oversubscription allows deploying more servers into a data center, and therefore reduces the number of data centers needed to be built. The cost savings potential of power oversubscription amounts to billions of dollars per year and is therefore of great importance to data center operators.

However, power oversubscription comes with a risk of overload during power peaks, and thus often comes with protective systems such as power capping. Power capping systems enable safe power oversubscription by preventing overload during power emergencies. Power capping actions include suspending low-priority tasks [18], throttling CPU voltage and frequency using techniques such as dynamic voltage and frequency scaling (DVFS) and running average power limit (RAPL) [8, 13, 25], or packing threads in a subset of available cores [17]. The action needs to be compatible with the workloads and meet their service-level objectives (SLOs). This, however, is challenging for clusters with throughput-oriented workloads co-located with latency-sensitive workloads on the same servers.

Throughput-oriented tasks represent an important class of computation workloads. Examples are web indexing, log processing, and machine learning model training. These workloads typically have deadlines on the order of hours for when the computation needs to be completed, making them good candidates for performance throttling when a cluster faces a power emergency due to power oversubscription. Nevertheless, missing the deadline can result in serious consequences such as lost revenue and diminished quality, thus making them unamenable to interruption.

Latency-sensitive workloads are a different class. They need to complete the requested computation on the order of milliseconds to seconds. A typical example is an application that handles user requests. High latencies result in bad user experience, eventually leading to loss of users and revenue. Unlike throughput-oriented pipelines, such tasks are not amenable to performance throttling. They often are considered high priority and need to be exempt from power capping.

In our data centers, throughput-oriented and latency-sensitive tasks are co-located on the same server to increase resource utilization [20]. This introduces a need for a fine-grained power capping mechanism that throttles the performance of throughput-oriented tasks to reduce server power usage while exempting high-priority latency-sensitive tasks.

This paper describes a simple, robust, and hardware-agnostic power capping system, Thunderbolt, to address these challenges. It throttles the CPU shares of throughput-oriented workloads to slow them down "just enough" to keep power under specified budget, while leaving latency-sensitive tasks unaffected. It has been deployed in large-scale production data centers.

To our knowledge, Thunderbolt is the first industry system that simultaneously achieves the following goals. All of these are important to scale out mission critical systems.

- A system architecture that enables oversubscription across large power domains. Power pooling and statistical multiplexing across machines in large power domains maximizes the potential for power oversubscription.
- Quality-of-service-aware, hardware-agnostic power throttling mechanism with wide applicability. Our system relies only on established Linux kernel features, and thus is hardware platform agnostic. This enables the flexibility of introducing a variety of platforms into data centers without compromising the effectiveness of power capping. Our task-level mechanisms allow differentiated quality-of-service (QoS). Specifically, Thunderbolt does not affect serving, latency-sensitive workloads co-located with throughput-oriented workloads on the same server, and has the ability to apply different CPU caps on workloads with different SLOs. The platform-agnostic and QoS-aware nature allows the system to be tailored to the requirements of a wide spectrum of hardware platforms and software applications.
- Power safety with minimized performance degradation. A two-threshold scheme with a randomized unthrottling/multiplicative decrease algorithm enables minimal performance impact while ensuring that a large amount of power can be shed to avoid power overload during emergencies.
- System availability in the face of power telemetry unavailability. Power telemetry availability has not drawn much attention in most previous power capping systems, but we found it to be the availability bottleneck in our system. Thunderbolt introduces a failover subsystem to maintain power safety guarantees even when power telemetry is unavailable.

We have deployed this system in multiple data centers over a period of two years. We have verified proper operation at scale and achieved oversubscription of 9–25% when none was previously possible. At such an oversubscription level, data centers run at high power efficiency and are close to the edge of exceeding their power limits. Power capping is expected to occur a few times a year. Section 7 has more details.

2 Background

Warehouse-sized data centers run very complex and diverse workloads and need a flexible power actuator to handle complicated application scenarios. Google recently published a power capping system [18] that intentionally suspends low-priority tasks which often results in task timeouts and failures. Such interruption is appropriate in certain situations; for instance, some tasks can tolerate occasional downtime but prefer to have consistent performance when they run, and prefer to be interrupted so they can be rescheduled somewhere else rather than being slowed down. However, for throughput-oriented workloads, this is not only wasteful of compute resources but is also disruptive.

Popular software frameworks like Hadoop [19], MillWheel [3], and TensorFlow [1] provide checkpointing functionality to allow tasks to handle failures gracefully. Checkpointing itself, however, incurs non-negligible cost and complexity. Users have to balance between the risk of failure and the overhead of runtime checkpointing. Even with checkpointing, some amount of work is wasted when a task is killed and restarted. For distributed computing that requires synchronization among workers (e.g., synchronized machine learning training), a killed task can easily become a straggler as others have to wait for it to make forward progress. Our system aims to provide a more graceful solution where traditional task killing or suspension is too costly.

Most previous studies control CPU power to affect overall machine power draw. Our system follows this practice, because CPU power draw is much higher than that of memory or storage components (e.g., flash and disk) on the commodity servers in our data centers.

Data center workloads also run at different priorities with varying QoS. It is highly desirable to reflect differentiated QoS even under power capping. Previous industry power capping systems, such as Dynamo [25] and CapMaestro [14], differentiate priorities at the machine level. They assign priorities to individual machines and build a global priority-aware control policy for all machines involved. To provide QoS

differentiation with Dynamo or CapMaestro, tasks with different capping priorities have to be scheduled on different machines. This conflicts with our requirement to run mixed-priority workloads on the same machine to improve resource utilization. In contrast, our approach is designed to provide QoS differentiation when workloads of different priorities run on the same machine.

From a scheduling perspective, our problem may look similar to the classic problem of scheduling latency-sensitive and throughput-oriented tasks on the same machine and optimizing for latency and throughput. However, it is a different form of the problem to which existing scheduling solutions do not directly apply. The constraining resource is power, which neither cluster scheduler nor local node scheduler can directly control or allocate. Instead we control power indirectly by controlling CPU usage. We treat power as a system output, measure it via power meters, and feed it back into the system to build a control loop. We have to care about the availability of power readings that are external signals. Violation of the power budget results in not performance degradation but high-stake physical failures (tripping circuit breakers) and immediate power loss to thousands of machines. Therefore a strong guarantee of power not exceeding the budget is the top priority, requiring fast response and a wide dynamic range of power control. Optimization for latency and throughput must not compromise this guarantee.

3 Terminology

To facilitate the explanation of our system, we define a few key terms summarized here for easy reference.

Thunderbolt. The power capping system as a whole, named after the resulting power curves that look like a thunderbolt (see Figure 5). The overall architecture is described in Section 4.

Load shaping. The "reactive capping" subsystem and closed-loop control policy using power signals for fine-grained power capping control. It is described in Section 4.1.2. It uses CPU bandwidth control (described below) as the node-level mechanism.

CPU bandwidth control. The node-level mechanism for load shaping. It leverages the CPU bandwidth control feature provided by Linux's completely fair scheduler to throttle the CPU usage of tasks. It is described in Section 4.1.1.

CPU jailing. The "proactive capping" backup subsystem that takes over when power signals are unavailable and load shaping cannot function. It includes an open-loop control policy of risk assessment and a node-level mechanism that makes use of Linux's CPU affinity features to limit machines' CPU utilization. It is described in Section 4.2.1.

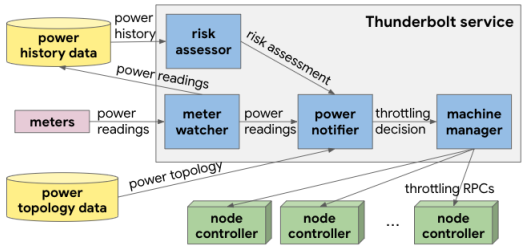


Figure 1: Software architecture of Thunderbolt.

4 Architecture and Implementation

Thunderbolt is capable of performing two types of end-to-end power capping actuation tailored to throughput-oriented workloads: a primary mechanism called reactive capping, and a failover mechanism called proactive capping. Reactive capping monitors real-time power signals read from power meters and reacts to high power measurements by throttling workloads. When power signals become unavailable, e.g., due to meter downtime, proactive capping takes over and assesses the risk of breaker trips. The assessment is based on factors such as power in the recent past and for how long the signals have been unavailable. If the risk is deemed high, it proactively throttles tasks.

The reactive capping system depends on power signals provided by power meters installed close to the protected power equipment, like circuit breakers. Meters are installed at every power "choke point" whose limit will be first reached as power draw increases. In our data centers the choke points are typically power distribution units (PDUs) or medium voltage power planes (MVPPs) [18]. This differs from the more widely adopted approach of collecting power measurements from individual compute nodes and aggregating at upper levels. Our approach has several advantages. It is simple. It avoids aggregation and the associated data quality issues such as time unalignment and partial collection failures. It also avoids the need to estimate power drawn by non-compute equipment, such as data center cooling, that does not provide power measurements.

Figure 1 illustrates the software architecture of Thunderbolt. The meter watcher module polls power readings from meters at a rate of one reading per second. It passes the readings to the power notifier module and also stores a copy in a power history datastore. The power notifier is a central module that implements the control logic of reactive and proactive capping. When power readings are available, it uses the readings for the reactive capping logic. When the readings are unavailable, it queries the risk assessor module for the proactive capping logic. The risk assessor uses the historical power information from the power history datastore to assess the risk of breaker trips. If either logic decides to cap, the power notifier will pass appropriate capping parameters to the ma-

chine manager module, which then sends remote procedure call (RPC) requests to the node controller of individual machines concurrently to reduce power. Important data about the power delivery topology, such as the protected power limits and the machines to be throttled under the power domain, are obtained from a power topology datastore.

The scale of a power domain can vary from a few megawatts, such as a PDU, to tens of megawatts, such as a MVPP. One instance of Thunderbolt is deployed for each protected power domain. The instance is replicated for fault tolerance. There are 4 replicas in a 2-leader, 2-follower configuration. Only the leader replicas can read power meters and issue power shedding RPCs. The node controller's power shedding RPC services are designed to be idempotent and can handle duplicate RPCs from different leader replicas. We require two identical leader replicas to ensure power shedding is available even during leader election periods. Followers take over when leaders become unavailable.

The architecture allows Thunderbolt to scale easily. When a new power domain is turned up, a new Thunderbolt instance can be deployed without affecting existing instances for other domains. When machines are added to or removed from a power domain, only the power topology data needs to be updated to include an up-to-date list of machines.

4.1 Primary subsystem: reactive capping

4.1.1 Node-level mechanism: CPU bandwidth control

CPU usage is a good indicator for the CPU power drawn by a running task. We use the CPU bandwidth control feature of the Linux completely fair scheduler (CFS) [21] to precisely control the CPU usage of tasks running on a node, in order to control the power drawn by the node.

Individual tasks run inside their own Linux control groups (cgroups). The Linux scheduler provides two parameters for a cgroup, namely *quota* and *period*. Quota controls the amount of CPU time the workload gets to run during a period and is replenished every period. Quota is shared and enforced by all logical CPUs in the system. The quota and period can be set for each cgroup and are typically specified at millisecond granularity. A separate (per cgroup and per logical CPU), cumulative *runtime_remaining* variable is kept inside the kernel. The cumulative (per logical CPU) *runtime_remaining* is consumed when a thread is running on the CPU. When it reaches zero, it attempts to draw from the per-cgroup quota pool. When the quota pool is empty, the running thread is descheduled and no thread in the same cgroup can run until quota is replenished at the beginning of the next period.

We track the historical CPU usage of all workloads running on the machine. During a capping event, every node in the power domain will receive an RPC to throttle throughput-oriented workloads. The RPC contains parameters describing how much the CPU usage of the tasks should be reduced (details in Section 4.1.2). The node controller that receives the RPC uses the historical CPU usage of all throughput oriented workloads to determine how much CPU time to throttle. The new quota and period values are then calculated and configured for each cgroup on the machine.

Different tasks have different cgroups, and we can achieve task-level QoS differentiation by adjusting their cgroup parameters. The Thunderbolt framework is capable of assigning different CPU throttling levels to different cgroups, with more restrictive levels to lower priority cgroups. In our cluster resource management systems, throughput-oriented tasks are typically assigned low priorities while latency-sensitive tasks are assigned high priorities. CPU throttling is applied only to cgroups of throughput-oriented tasks, exempting cgroups of latency-sensitive tasks. This is appropriate for our production environment, where a significant portion of total CPU resources is consumed by throughput-oriented tasks (also known as batch tasks [20]), and it is undesirable to throttle business-critical latency-sensitive tasks. Exempting all latency-sensitive tasks comes with a caution about non-sheddable power, which is explained below. Kernel threads are also exempt from throttling and their CPU usage is very low compared to regular tasks.

Non-sheddable power, the lower bound of the power control range, is an important consideration for power oversubscription and capping. With our implementation of Thunderbolt, non-sheddable power can be attributed to CPU usage by exempt tasks, machine idle power, and other uncontrolled power users such as cooling equipment. We deliberately set the oversubscription level so that non-sheddable power does not exceed the protected power limits. We run continuous monitoring and rigorous analysis to predict and alert on the portion of non-sheddable power in our data centers. In an unlikely event when high non-sheddable power is predicted in a cluster, site operators can leverage global load balancing to redirect traffic of latency-sensitive tasks elsewhere to offset the risk.

The relationship between throttling levels and power draw is nonlinear and workload dependent, therefore we always use CPU bandwidth control in conjunction with power metering and negative feedback to ensure expected power reduction is achieved. The feedback loop is described in detail in Section 4.1.2.

Characteristics of CPU bandwidth control. CPU bandwidth control has two important properties:

Platform-agnostic. CPU bandwidth control is a pure software feature supported by the upstream Linux kernel. It can be switched on for almost any new platform with minimal additional effort.

Task-level control. CPU bandwidth control is at the task (cgroup) level. Specifically, tasks of varying priorities are co-located on the same server and can even run on the same physical core. CPU bandwidth control has the

Table 1: Comparison between CPU bandwidth control, DVFS, and RAPL for power limiting.

	CPU bandwidth control	DVFS	RAPL
Response time	1 ms	100 μs	100 μs
Spatial granularity	cgroup	Physical core	Processor socket
Power feedback control	Requires external	Requires external	Processor built-in
Mechanism	Pure software	Requires hardware support	Requires hardware support

required fine-grained visibility and control to provide the differentiated QoS.

These properties make CPU bandwidth control a good fit for our needs. We have also considered other popular hardware-based alternatives, in particular dynamic voltage and frequency scaling (DVFS) and Intel's running average power limit (RAPL). Below we compare and discuss CPU bandwidth control and the two alternatives, and explain why, despite the merits of the two alternatives, we do not adopt them for Thunderbolt.

We summarize several attributes of CPU bandwidth control, DVFS, and RAPL in Table 1. Given their nature of hardware control, DVFS and RAPL both have faster power response times than bandwidth control. In practice, however, we find that the longer response time of CPU bandwidth control is still fast enough to be an effective load shedding mechanism for safe power oversubscription (see Section 6).

CPU bandwidth control vs RAPL: RAPL is available only on Intel platforms. More importantly, the power limit can only be set on a per socket basis, which means it does not provide task-level control granularity. Alternative approaches are possible to achieve differentiated task QoS using RAPL if additional support is added to the node controller. For instance, tasks with different QoS may be scheduled on different sockets. Apart from the extra complexity, such a scheduling constraint has a disadvantage of limiting CPU resource overcommitment opportunity, which is undesirable for our cluster scheduler [22].

CPU bandwidth control vs DVFS: DVFS is available on most modern high-performance platforms, bringing its compatibility close to CPU bandwidth control. However, it may also have problems supporting task-level control. For example, per-core DVFS is supported by Intel only for Haswell and later generations, and it is not supported by some non-x86 vendors. In terms of power control and performance impact, as we will show in Section 5, DVFS is incapable of throttling down to very low power levels but it has better power efficiency than bandwidth control.

Operational factors. The platform-agnostic nature of CPU bandwidth control is vital to new platform introductions. Even if a new microarchitecture supports fine-grained DVFS, driver support for new platforms often have issues that require extra work. More importantly, per-task DVFS setting is not supported by the upstream Linux kernel. It is also not rare to find chip errata that require workarounds. Using CPU bandwidth control as either the main throttling mechanism or as a fallback mechanism removes these uncertainties in the critical path. It makes us more comfortable about scaling up our data centers with heterogeneous processor microarchitectures.

Overall we consider CPU bandwidth control essential to the success of Thunderbolt. In the future DVFS can be added as a node-level optimization. When Thunderbolt was first deployed, per-task DVFS setting was not available in our Linux kernel. We have recently added per-task DVFS support to the Linux kernel to enable additional trade-offs between performance and efficiency on Intel servers. The same kernel mechanism can be used for power throttling.

4.1.2 Control policy: load shaping

The load shaping control policy determines when and how much the actuator (CPU bandwidth control) should throttle CPU usage in order to control power.

Formally, the power draw of a power domain can be written as

$$p(t) = \sum_{i=1}^{N} f_i(c_i(t) + u_i(t)) + n(t) \quad (1)$$

where t is (discrete) time, p is the total power draw, N is the number of machines, f_i is the power drawn by machine i as a monotonic function of the normalized machine CPU utilization (in the range of [0, 1]), c_i is the CPU used by controllable tasks, u_i is the uncontrollable CPU used by exempt tasks and the Linux kernel, and n is the power drawn by non-machine equipment. Our goal is to cap c_i so that $p < l$ for a power limit l. Preventing overload ($p > l$) is the top priority, while keeping p close to l when $p < l$ is also desirable for efficiency.

We use a *randomized unthrottling/multiplicative decrease* (RUMD) algorithm. If $p(t) > l$, then we apply a cap for the CPU usage of each controllable task. The cap is equal to the task's previous usage multiplied by a *multiplier*, m, in the

range of (0, 1). Then the power draw at the next time step is

$$\begin{aligned} p(t+1) &= \sum_{i=1}^{N} f_i(c_i(t+1)+u_i(t+1))+n(t+1) \\ &\leq \sum_{i=1}^{N} f_i(mc_i(t)+u_i(t+1))+n(t+1) \end{aligned} \tag{2}$$

This cap is updated every second, and c_i decreases exponentially with time, until $p < l$. Note that, because of the u_i and n terms, there is no guarantee that $p(t+1) < p(t)$. Nevertheless, as explained in Section 4.1.1, in practice we ensure with high confidence that non-sheddable power is less than the power limit, that is,

$$\sum_{i=1}^{N} f_i(u_i(t))+n(t) < l, \forall t \tag{3}$$

Therefore power will eventually be reduced below the limit.

The system works on a time scale of seconds: power measurements are read once per second, and throttling parameters are updated every second. This is because the typical end-to-end response time is 1–2 seconds from a high power draw to power being sufficiently reduced by throttling. This is mostly attributed to metering delays. We conservatively budget 5 seconds to account for occasionally longer metering delays and network tail latency.

Throttling stops when p decreases to be below l. To avoid fast power surges, throttling should stop in a progressive manner. We do this by removing the CPU cap on a random portion of machines every second. For instance, if it is configured to completely unthrottle all machines in 5 seconds, then a random non-overlapping set of 20% of machines will be unthrottled every second. Alternatively, one may progressively lift the cap in an additive manner for each machine at the same time, leading to an *additive increase/multiplicative decrease* (AIMD) algorithm [5]. We choose a randomized unthrottling scheme instead of AIMD because it is simpler (no need for an additive increase parameter), and AIMD's "fairness" property (machines converging to having the same CPU utilization) is not required for our system, as long as randomization avoids any machine from being disproportionally impacted.

Similar to AIMD, our RUMD algorithm also has the desirable partial distributedness property. The central policy controller requires no detailed system states, such as the CPU usage and task distribution of each machine, other than the total power. The distributed node controllers can make independent decisions based solely on a few parameters that the policy controller sends to all node controllers.

The result of the RUMD algorithm is a power curve oscillating around the capping limit in a sawtooth-like pattern, as can be seen in Section 6.2.

Implementation details. Here we give some details about our implementation of the RUMD algorithm. In particular, we explain how we balance two competing properties, responsiveness for power safety and efficiency for minimizing performance impact, by maintaining two capping thresholds, one high and one low. The *high threshold*, placed close to the protected power limit, is associated with a *hard multiplier* close to 0 in order to quickly reduce power for safety. The *low threshold*, placed with a larger margin from the protected limit, is associated with a *soft multiplier* for gentle throttling.

We start by explaining the high threshold for power safety. Our end-to-end response time budget is 5 seconds. In 5 seconds, we have observed that power in a nearly full cluster will increase by no more than 2% of the protected equipment limit. Therefore we place the high threshold at 98% of the limit. The hard multiplier associated with this threshold is set to be close to 0 for a quick reduction of a large amount of power. This is because the only strong power guarantee is non-sheddable power being less than the limit (Equation 3), and thus sheddable power has to be reduced to nearly zero quickly to guarantee responsiveness and safety.

It is worth noting that most circuit breakers do not immediately trip when their rated power limit is reached. They may tolerate a few seconds to tens of minutes of power overload [9]. In theory we may make use of this time buffer and set the capping threshold at the power limit. However, how long a breaker can sustain a power overload depends on many factors, such as the design of the breaker, the magnitude of the overload, and ambient temperature [9], and is thus hard to predict. Power overload also decreases the equipment's lifetime. Therefore we choose to place the high threshold 2% below the power limit to avoid tapping into the overload region.

We do not reduce the CPU cap of a task below a minimum value (0.01) because the quota value in CPU bandwidth control has to be greater than zero. This has a production implication: when continuous throttling is applied long enough, affected tasks will eventually converge to the minimum CPU share. In this case, while all affected tasks cannot make meaningful progress and power will be low, some tasks can still respond to health checks and survive. Because of this, task failures due to continuous throttling are expected to be fewer than failures caused by completely suspending tasks, as can be seen in Section 6.2.

The hard multiplier close to zero, while being responsive and safe, is not efficient for utilizing the power budget because it leads to power oscillation with a large amplitude. Therefore we introduce the low threshold associated with the soft multiplier. The soft multiplier is close to 1 to improve efficiency at the cost of responsiveness, and the low threshold is placed below the high threshold to allow the longer response time.

We further optimize our design by not activating the low threshold until throttling is triggered, and deactivating it after throttling has not been active for a while. This way power is allowed to reach the range between the two thresholds without throttling.

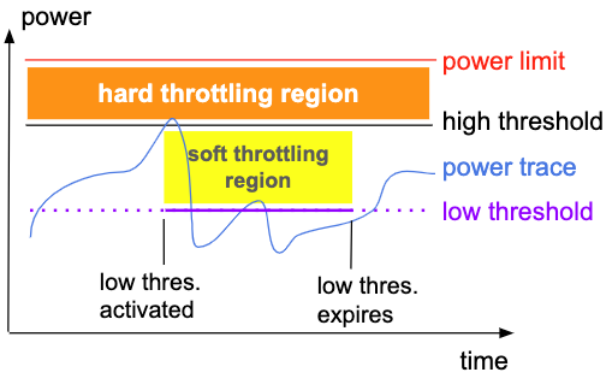


Figure 2: Load shaping power control.

Table 2: Load shaping parameters used in production.

Load shaping parameter	Value
High threshold	98% of protected limit
Low threshold	96% of protected limit
Hard multiplier	0.01
Soft multiplier	0.75
Low threshold expiration	5 minutes
Throttling timeout	1–20 seconds

Randomized unthrottling is implemented by assigning to each machine a random throttling timeout in a range. A random timeout is included in the throttling RPCs and sent to each machine every second to refresh its timeout. When power is below the capping threshold, the machines will stop receiving the RPCs and will unthrottle after the last received timeout has passed. We choose a repeatedly-refreshed timeout instead of a stop-throttling RPC because stop-throttling RPCs may be dropped or even never reach some machines if the network becomes partitioned.

Figure 2 illustrates the power trace in a typical throttling scenario. Table 2 lists the parameters we use in production.

4.2 Failover subsystem: proactive capping

The feedback control of reactive capping relies on power meters. However, power meters and the facility network connecting the meters to the production network are not always available. On average, individual meters and facility network have about 99.9% availability in our data centers, and it varies by location. Transient network issues can cause seconds to minutes of power signal interruption, while meter downtime can be days before the meter gets repaired.

Without power signals, it is not straightforward to use CPU bandwidth control for an open-loop control. We have built models to map machine power utilization to CPU utilization, so we may distribute the power domain's total power budget to individual machines and translate a machine's power budget to a CPU budget. However, it would require a sophisticated algorithm to allocate the machine's CPU budget among individual tasks while respecting the tasks' QoS difference. Instead of introducing a complex algorithm, we implement a simple mechanism, CPU jailing, that specifies a total CPU budget for a machine and leverages the Linux CFS scheduler to provide task QoS differentiation (although the differentiation is relaxed compared to when meter is available, which is explained further below). In a nutshell, CPU jailing is coarser-grained than CPU bandwidth control, but much easier to reason about when power signals are unavailable.

DVFS or RAPL, where supported, may also be used for proactive capping because we only need machine-level control. However, we favor the platform-independent CPU jailing for the same reasons as we favor the platform independence of CPU bandwidth control.

We have also considered collecting power signals from secondary sources, such as the machines' power supply units, or from power models. However, we found that the data quality of the sensors and the accuracy of the models for some hardware do not meet our production requirements.

4.2.1 Node-level mechanism: CPU jailing

CPU jailing masks out ("jails") a certain number of logical CPUs from tasks' runnable CPU affinity [15] to cap total machine power. We refer to the portion of jailed CPUs as *jailing fraction*, denoted by J. CFS will maintain proportional fairness among tasks on the remaining available logical CPUs. Each jailing request comes with a timeout that can be renewed. Once jailing expires, previously masked CPUs will immediately become available to all tasks.

CPU jailing immediately caps peak power draw as it effectively limits maximum CPU utilization to $(1-J)$ on every single machine. It sets an upper bound for power draw, allowing safe operation for an extended time without power signals. Because of increased idleness, jailed CPUs have a higher chance of entering deep sleep to further reduce machine power.

The jailing fraction is uniformly applied to individual machines, regardless of their CPU utilization. Consequently, machines with low utilization are less impacted than highly utilized machines. As an extreme example, CPU jailing might not affect tasks at all on machines with utilization well below $(1-J)$.

Certain privileged processes, such as critical system daemons, are explicitly exempt (i.e., they can still run on jailed CPUs). The rationale is that their CPU usage is very low compared to regular tasks but the consequences of them being CPU starved can be devastating (e.g., machine cannot function correctly). A side effect of exemption is that it puts some sporadic usage on the jailed CPUs and occasionally prevents

them from entering deep sleep state.

A main disadvantage of CPU jailing is the relaxed QoS differentiation. For example, the latency of a serving task can be severely affected when too many cores are jailed. Although this effect is attenuated by the fact that latency-sensitive tasks run at higher priorities and can preempt lower-priority throughput-oriented tasks during CPU resource contention, CPU jailing is less favorable than CPU bandwidth control and is only employed where load shaping is not applicable.

Technically, one may achieve strict QoS differentiation by applying CPU jailing to only throughput-oriented tasks while exempting latency-sensitive ones. However, doing so without power signals is intangible in practice. If latency-sensitive tasks are exempt from CPU jailing, the only strong guarantee we have about power is that non-sheddable power does not exceed power limit (Equation 3). In this situation, guaranteeing power safety would require not running throughput-oriented tasks at all, which we cannot afford.

Determining jailing fraction J. A proper jailing fraction J can be determined from two factors: the relation between CPU utilization and power utilization, and power oversubscription ratio.

For power safety, we need to ensure power is reduced to a safe level after a certain fraction of CPUs are jailed. This value of J can be calculated from the power oversubscription ratio and the CPU utilization-power utilization relation of the given collection of hardware in the power domain, as follows:

$$J = 1 - U_{cpu} = 1 - g_{power \to cpu}(\frac{1}{1+r}) \tag{4}$$

In the formula, U_{cpu} is the highest allowed CPU utilization (normalized to the total CPU capacity), $g_{power \to cpu}$ is a function to convert power utilization (normalized to the theoretical total peak power) to CPU utilization, and r is the oversubscription ratio defined by the extra oversubscribed power capacity as a fraction of the nominal capacity. $1/(1+r)$ gives the maximum safe power utilization, which can be converted to U_{cpu} given that the CPU utilization-power utilization relation is monotonic. A greater r leads to smaller allowed power utilization and smaller U_{cpu}, which in turn leads to greater J.

In production, we set J to 20%–50% depending on a cluster's workloads and risk profiles. This is a deliberate trade-off between performance SLO and power oversubscription opportunity.

4.2.2 Control policy: risk assessment of power signal unavailability

As a fallback approach, CPU jailing is triggered when we lose power measurements from the meters and the risk of power overloading is high. The risk is determined by two factors, predicted power draw and meter unavailability duration. Higher predicted power draw and longer meter unavailability means higher risk. The end-to-end delay from risk assessment to power reduction is typically 1–2 seconds, similar to load shaping. In our implementation, we use a simple and conservative probabilistic model to estimate the probability of power reaching the protected equipment limit during certain meter downtime given the power draw of the recent past. CPU jailing is triggered if the probability is high due to high recent power draw and long enough downtime. Our conservative model favors low false negatives (i.e., CPU jailing is triggered when overload would have happened without it) at the cost of relatively high false positives (i.e., CPU jailing is triggered even when it does not have to). This is appropriate because power safety is our top priority and power reading unavailability is infrequent. The probabilistic model is not the focus of this paper, but one can freely use any model that estimates the risk from any available data and plug it in here.

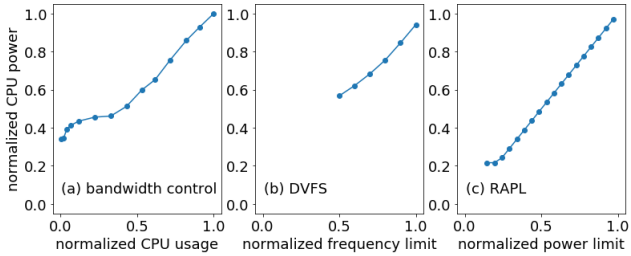


Figure 3: CPU power response to bandwidth control, DVFS, and RAPL.

5 Evaluation Results at the Node Level

Before discussing data center-level aggregated data, we show two examples of node-level data from experiments performed on an Intel Skylake CPU.

CPU power and set point. To quantify the effectiveness of CPU bandwidth control, DVFS, and RAPL to control power, we measure total CPU power under various set points of the three knobs. We ran Intel's "power virus" workload [7] that stresses the CPU and the memory to maximize power draw. We then separately used CPU bandwidth control, DVFS, and RAPL to limit CPU power and compared the results, which are shown in Figure 3. CPU power is normalized to the highest power observed when none of the power management mechanisms are enabled.

Figure 3(a) shows that, with CPU bandwidth control, we are able to reduce CPU power to 0.34 due to significant deep sleep state residency from bandwidth control.

In comparison, Figure 3(b) shows that with DVFS, power draw is still relatively high at 0.57 when the lowest frequency limit is applied. The frequency limit is normalized to the base frequency of the processor.

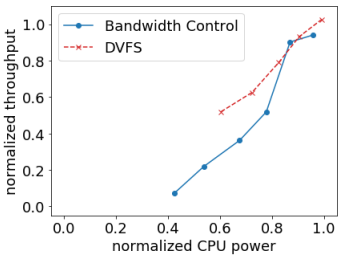


Figure 4: Throughput of a video transcoding task as a function of CPU power under bandwidth control and DVFS.

Figure 3(c) shows RAPL has the widest power reduction range among the three. It is able to reduce power to 0.22. However, we noticed system management tasks were sluggish to respond when RAPL approached the lowest power limits, which suggests higher machine timeout risks if these limits were actually used in practice. By contrast, CPU bandwidth control used in our system only throttles throughput-oriented tasks and the system management tasks are not affected. Thanks to its built-in feedback loop, RAPL is fairly accurate in achieving the provided power budget [26]. RAPL's predictability is an advantage over DVFS or CPU bandwidth control.

CPU power and throughput. In this experiment, we run a throughput-oriented video transcoding task under various set points of CPU bandwidth control and DVFS, and measure CPU power and task throughput. This gives us information about the throughput impact of the two mechanisms under a power budget. Throughput is calculated as the reciprocal of the wall clock time of completing the task, normalized to the throughput where no power throttling is applied. CPU power is normalized to the highest power observed when power virus is run and no power throttling is applied (matching Figure 3).

Results are shown in Figure 4. Throughput is only mildly affected when power is greater than 0.85 for both bandwidth control and DVFS. Possibly memory bandwidth, rather than CPU bandwidth, is the bottleneck in this region. Throughput drops notably as power drops below 0.85 for both mechanisms, but DVFS has higher throughput than bandwidth control under the same power. Therefore, DVFS is more power efficient than bandwidth control. However, in terms of power control dynamic range, DVFS can only reduce power by 40% when the lowest frequency limit is applied, whereas bandwidth control is capable of nearly 60% power reduction. This is consistent with the power virus result in Figure 3. Load shaping events happen infrequently in our data centers, thus power efficiency is not a major factor for our use case.

6 Evaluation Results at Data Center Scale

To characterize the system at scale, we performed experiments in clusters comprising tens of thousands of machines running

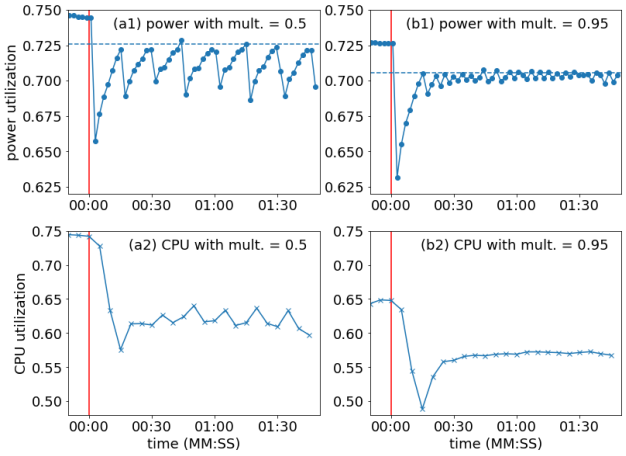


Figure 5: Typical load shaping patterns. (a1) and (a2) show the normalized power and CPU utilization of a load-shaped power domain, with 0.01 hard multiplier and 0.5 soft multiplier. (b1) and (b2) show similar data for the same power domain but with 0.01 hard multiplier and 0.95 soft multiplier. The blue horizontal dashed lines are low power thresholds associated with the soft multipliers. The red vertical lines mark the start of load shaping. (The power and CPU readings are not exactly time-aligned due to sampling delays.)

diverse production workloads in our data centers. Throttling was manually triggered with various combinations of parameters. Power data is collected from data center power meters, which is the same data that Thunderbolt also uses. Power measurement data is normalized to the power domain's equipment limit.

Other metrics are sampled from individual machines and aggregated at the same power domain level corresponding to the power readings. Machine metrics such as CPU usage are normalized to the total capacity of all machines in the power domain unless specified otherwise. Task failures are normalized to the total number of affected tasks.

Latency data are collected from low-level storage services that read and write Linux files and support Google's distributed file system. They are critical services widely deployed in our data centers, running at high priorities and thus exempt from load shaping. They are not exempt from CPU jailing but have high priority to access the remaining CPUs.

6.1 Load shaping in typical scenarios

In this experiment, we picked a production cluster that is dominated by throughput-oriented workloads to test the typical behavior of load shaping. Load shaping was triggered by manually lowering the high power threshold to be just below the ongoing power draw of a power domain.

Power and CPU usage patterns. Figure 5(a1) shows a typi-

Table 3: Load shaping duration, task failure fraction, and 99%-ile read latency of storage services under different scenarios.

	Duration	Failure fraction	Latency
Baseline	25 min.	0.00002	79 ms
0.95 soft mult.	5 min.	0.00000	79 ms
0.75 soft mult.	10 min.	0.00003	80 ms
0.5 soft mult.	5 min.	0.00007	78 ms

cal load shaping pattern of power oscillating around the low threshold. Seconds after throttling is triggered, power is reduced by a large margin because of the hard multiplier. Meanwhile the low threshold is activated. Throttling is gradually lifted as power drops below the low threshold, and power goes back up until it reaches the low threshold. Then power is reduced again, but by a smaller margin because of the soft multiplier. The process continues as throttling is turned on and off repeatedly, resulting in power oscillating around the low threshold. Figure 5(b1), as compared to (a1), shows a soft multiplier closer to 1.0 leads to oscillations of a smaller amplitude, as expected. The response time from load shaping triggering to significant power reduction is about 2 seconds.

Figure 5(a2) and (b2) show the CPU utilization corresponding to (a1) and (b1) respectively. At the shown CPU utilization level, about 0.1 reduction of CPU utilization is needed to reduce 0.02 of power.

Task failures. While tasks are slowed down, we want to ensure that most of them do not fail because of CPU starvation or unexpected side effects. Table 3 shows task failure fractions (the number of failed tasks normalized to the total number of affected tasks) of the same power domain under load shaping with various soft multipliers. "Baseline" indicates no throttling and serves as the baseline for comparison. All load shaping events have a hard multiplier of 0.01 (not shown in the table) while the soft multiplier varies from 0.5 to 0.95. Clearly load shaping does not cause noticeably more failures. The failure fraction remains low compared to the baseline.

Latencies. To assess load shaping's effect on the latencies of latency-sensitive tasks, we inspect the tail 99%-ile read latency of latency-sensitive storage services, shown in Table 3. As expected, the latency is not affected by load shaping because the tasks are exempt from the mechanism.

Differentiation of QoS. To test Thunderbolt's ability to differentiate QoS, we classified tasks into two groups based on their priority, and load-shaped the low-priority group while exempting the high-priority group. Figure 6 shows the total power draw and CPU usage of the two groups of tasks, during the event. The CPU usage of the shaped and the exempt group is reduced by about 0.1 and 0.03, respectively. The exempt group is indirectly affected because the tasks in the



Figure 6: Power and CPU utilization during a load shaping event with multiplier 0.1 that directly affects a subset of tasks. The red vertical line marks the start of the event. The CPU reduction of the load-shaped tasks are more prominent than that of the exempt tasks. The exempt tasks are indirectly affected because of their interaction with the shaped tasks. (The power and CPU readings are not exactly time-aligned due to sampling delays.)

two groups are production tasks with complex interactions. One of such interactions is that a high-priority controller task in the exempt group coordinates low-priority workers in the shaped group, and the controller task has less work to do and consumes less CPU when the workers are throttled. Nevertheless, the ability of load shaping to differentiate tasks is evident.

6.2 Load shaping pushed to the limit

In typical scenarios, as demonstrated in Section 6.1, load shaping reduces power to a safe level just below the threshold and allows power to oscillate around it. However, in extreme cases where power stays above the threshold, the system will need to continuously reduce tasks' CPU usage, eventually to the preset minimum value. The affected tasks will essentially be stopped and make no forward progress. For example, power may remain high after throttling is triggered because new compute-intense tasks are continuously scheduled, or many high-priority tasks exempt from the mechanism spike in their CPU usage. In such cases it is the right trade-off to stop the low-priority tasks in order to prevent power overloading.

To test the behavior of load shaping in such extreme scenarios, we picked a cluster with some low-priority, non-production, throughput-oriented workloads and applied a multiplier continuously to those tasks. (Most of the tasks in that cluster are high-priority, which we exempt from this test thanks to load shaping's ability to differentiate tasks.) We com-

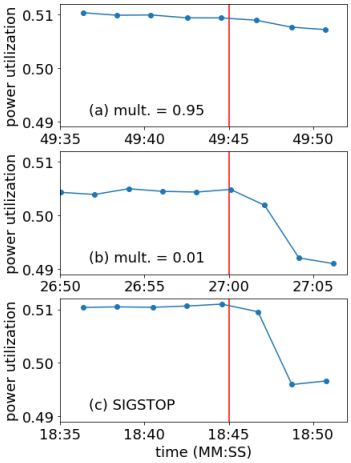


Figure 7: Power responsiveness of continuous throttling and of SIGSTOP. (a) and (b) are throttling with a multiplier of 0.95 and 0.01, respectively. (c) is SIGSTOP. The red vertical lines mark the start of throttling and SIGSTOP.

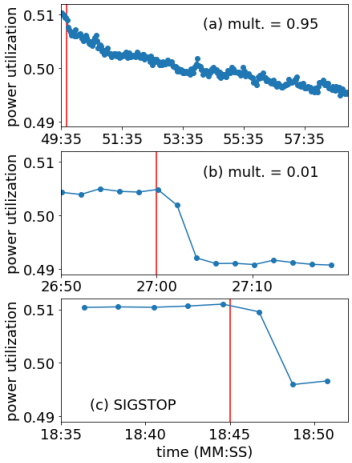


Figure 8: Power reduction by continuous throttling and by SIGSTOP. (a) and (b) are throttling with a multiplier of 0.95 and 0.01, respectively. (c) is SIGSTOP. The red vertical lines mark the start of throttling and SIGSTOP. The x axes of the sub-figures are scaled differently because the power reduction happens at different time scales.

pared continuous throttling to explicitly stopping the tasks by sending them a SIGSTOP signal followed by a SIGCONT signal after 60–75 seconds.

Power responsiveness and range of control. Figure 7 shows the power responsiveness of continuous throttling and of SIGSTOP. Here power is reduced noticeably in 2 seconds. This is true for all the tested multipliers as well as for SIGSTOP. In 4 seconds, about 3% of power is shed by throttling with a 0.01 multiplier and by SIGSTOP, and 0.5% by throttling with a 0.95 multiplier, respectively.

If throttling is applied continuously, we expect tasks to eventually have close-to-zero CPU shares and we achieve similar power reduction as SIGSTOP. This is indeed true. Figure 8 compares the power reduction by continuous throttling with two multipliers, and compares them to SIGSTOP. It plots the same data as Figure 7 but on a larger time scale to show the power reduction. (Note that the x axes of the sub-figures are scaled differently because the power reduction happens at different time scales.) Power is reduced at a slower pace with a multiplier closer to 1, but given enough time it is eventually reduced by an amount similar to SIGSTOP (about 0.015) regardless of multiplier. This is expected, because the cumulative effect of applying any multiplier between 0 and 1 should eventually converge to CPU shares that are close to zero. This also implies that the selection of the multiplier does not affect the effectiveness of power reduction in terms of sheddable power. The selection of the multiplier does affect responsiveness, however, which is important when power spikes need to be throttled quickly.

Task failures. Table 4 lists the task failure fractions during the test periods of continuous throttling and SIGSTOP. "Baseline" indicates no throttling or SIGSTOP and serves as baseline for comparison. Throttling with a 0.95 multiplier has mild effect on failure fraction and can be continuously applied to tasks for longer time (20 minutes here). Both throttling with a 0.01 multiplier and SIGSTOP were only performed for a short period of time (2 minutes), but they caused skyrocketed failure fraction by one to two orders of magnitude. The failures are mostly attributed to tasks being terminated because they fail to respond to health checks. The increased failure fraction of continuous throttling with a 0.01 multiplier is contrasted with the low failure fractions of load shaping in Table 3. Those load shaping events in Table 3 had a 0.01 hard multiplier in effect only for a few seconds, because the hard multiplier was progressively lifted in seconds after power drops below the high power threshold. The failure fraction of continuous throttling with a 0.01 multiplier is one order of magnitude lower than that of SIGSTOP because the throttled tasks still have a minimum CPU share, and some of them can respond

Table 4: Power shedding duration, task failure fraction, and 99%-ile read latency of storage services under different scenarios.

	Duration	Failure fraction	Latency
Baseline	15 min.	0.0007	126 ms
0.95 mult.	20 min.	0.0007	122 ms
0.01 mult.	2 min.	0.003	125 ms
SIGSTOP	2 min.	0.06	135 ms

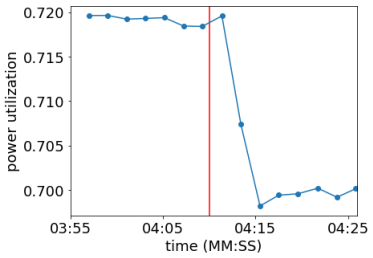


Figure 9: Responsiveness and power reduction of CPU jailing with 20% jailing fraction. Power is reduced by 0.02 in 5 seconds when the power domain's CPU utilization is about 60% (not shown in the figure).

Table 5: Effect of 20% CPU jailing on machine CPU utilization.

	Duration	Machine CPU utilization		
		50%ile	95%ile	99%ile
Baseline	60 min.	0.58	0.80	0.94
CPU jailing	55 min.	0.55	0.69	0.75

to health checks and survive.

Latencies. Table 4 shows the tail 99%-ile read latency of latency-sensitive storage services. As expected, the latency is not notably affected by either load shaping or SIGSTOP, both of which are not applied to those services.

6.3 CPU jailing

For this experiment of CPU jailing, we picked the same production cluster as in Section 6.1, which is dominated by throughput-oriented workloads. We manually performed CPU jailing with a 0.2 jailing fraction, denoted by "20% CPU jailing", and collected data for power, CPU usage, CPU cores in deep sleep states, task failures, and latencies. The same types of data were collected during a period before the CPU jailing event; those data will serve as the baseline for comparison.

Power responsiveness. For the purpose of a failover mechanism, response time is not a concern in most cases, except for the corner case where meter signals are lost while load shaping is, or very close to being, active. Nevertheless, Figure 9 shows that power utilization is reduced by 0.02 (from 0.72 to 0.70) in 5 seconds under 20% CPU jailing. The power reduction is relatively small, because most machines had lower than 80% CPU utilization even before 20% jailing was applied to them. This can be seen in Table 5, discussed further below.

CPU usage. CPU jailing affects machines with high CPU utilization more than those with low CPU utilization. This is evident from Table 5. The median machine CPU utilization without CPU jailing is 0.58, and it is only mildly affected by 20% CPU jailing that limits available machine CPU capacity to 80%. In contrast, the 99%-ile and 95%-ile machine CPU utilizations, which are close to or higher than 80%, are reduced significantly during CPU jailing.

While CPU jailing is a pure software mechanism, it can get extra benefits with hardware support that puts idle cores in power-saving states. In our experiment with 20% CPU jailing, 5% of affected CPU cores entered deep sleep states (C6/C7 states) as compared to 1% of cores without jailing. Noticeably, although 20% of cores are jailed, the portion of deep-sleep cores is always less than 20% due to processes exempt or unaffected by the mechanism.

Table 6: Task failure fraction and 99%-ile read latency of storage services under 20% CPU jailing.

	Duration	Failure fraction	Latency
Baseline	60 min.	0.00003	79 ms
CPU jailing	55 min.	0.00002	86 ms

Task failures and latencies. Table 6 shows the task failure fraction and the 99%-ile read latency of storage services of a power domain in a 20% CPU jailing event. There is no notable difference in failure fraction and latency compared to the baseline. Both latencies are far below our SLO. However, in a separate experiment of 80% CPU jailing we observed an order of magnitude higher latency (not shown in the table), which is not surprising because severe CPU contention is expected with such heavy jailing.

7 Deployment at Scale and Benefits

Thunderbolt has been deployed at scale in our logs processing clusters and has enabled 9%–25% power oversubscription relative to the nominal capacity, depending on the power delivery architecture. The oversubscription is determined by an SLO with the clusters' stakeholders about the expected occurrence frequency of throttling events under realistic worst conditions. Other throughput-oriented clusters, such as web indexing, are also in scope of more aggressive power oversubscription with Thunderbolt.

Logs processing workloads are mostly throughput-oriented and continuously running. Resources are provisioned to accommodate worst-case daily throughput demands, and any disruptive power capping actuation on the worst day is a waste of the resources and cancels the benefits of aggressive power oversubscription. Thunderbolt, by gently throttling computation, distributes the actual work throughout the day, gracefully allowing throughput to be conserved. Despite that most workloads are throughput-oriented, there are still critical latency-sensitive workloads such as low-level storage services, and

therefore QoS differentiation is important.

The reactive capping mechanism has been activated three times by exceptionally high power draw in three production clusters in the first 130 days of year 2020. The proactive capping mechanism has been activated two times by power telemetry unavailability in two high-power production clusters in the same period of time. Such incidents could have resulted in tripping data center breakers without the protection from the power capping system. The activation events went unnoticed by stakeholders, with negligible adverse effect on production.

8 Challenges and Future Work

Thunderbolt, implemented as described in this paper, is suitable for our production clusters running a mix of throughput-oriented and latency-sensitive workloads. Those clusters have a sizable portion of power drawn by throughput-oriented tasks, and a stable usage pattern of latency-sensitive workloads. Therefore, we are able to set an appropriate oversubscription level with high confidence that non-sheddable power will not pose a risk, and that CPU jailing will not starve latency-sensitive tasks. Nevertheless, the Thunderbolt framework is flexible enough for extension and optimization to accommodate clusters of different workload patterns. Here we discuss some directions and challenges.

Our implementation exempts all high-priority latency-sensitive workloads from load shaping but this is not always required. In clusters where latency-sensitive workloads may use too much power, one could further break them down into multiple priority buckets and throttle them as appropriate under their SLOs. Doing so in practice is a challenge as latency-sensitive tasks are generally not amenable to CPU throttling. It will likely require a co-design of throttling policy, SLO, and software infrastructure. For example, one could have an SLO that permits affecting the latencies for a small fraction of time, and design the workload and software infrastructure to respond to high latency properly. For cloud data centers where the infrastructure owner has limited control over the workloads, cloud providers may carefully design service-level agreements (SLAs) to allow throttling "abusive" behaviors, and possibly use price incentives to encourage "good" behaviors.

Thunderbolt sheds power by controlling CPU usage. This may not be effective if the majority of power is used by non-CPU components, such as hardware accelerators. While hardware support is needed to effectively throttle such components, the Thunderbolt software architecture and control policies of load shaping and proactive capping can be adapted to control additional hardware throttling knobs. QoS differentiation will depend on the control granularity of the hardware. For example, if an accelerator supports per-chip throttling and a chip is used by one task at a time, then task-level QoS differentiation is possible.

While proactive capping addresses the availability bottleneck of power telemetry unavailability, it may become a limiting factor for power oversubscription. We have to set the jailing fraction conservatively (i.e., it may be set greater than necessary) for the open-loop control to be safe. For clusters with a high portion of latency-sensitive tasks, only a small jailing fraction may be feasible, leading to a small oversubscription. To increase oversubscription for those clusters, it may be worth investing in building a reliable secondary source of power signals, either from rack- or machine-level power sensors or from machine learning models that map resource usage to power, so that closed-loop control is still functional when the primary source, data center power meters, is unavailable. Proactive capping may be used as the last resort when both the primary and the secondary sources are unavailable.

Thunderbolt is a *reactive* system (not to be confused with "reactive capping" defined in this paper), in the sense that it reacts to riskily high data center power that is present (in the case of reactive capping) or expected (in the case of proactive capping). A more *proactive* approach, such as power-aware job scheduling and admission control, may actively balance load to avoid riskily high data center power via scheduling rather than throttling. Job scheduling and admission control are largely orthogonal and complementary to Thunderbolt and are valuable candidates for future work.

9 Related Work

This work has a similar architecture as Google's power capping for medium-voltage power planes (MVPPs) [18]. It shares many advantages of the MVPP power capping, such as fast response, priority- and QoS-awareness, platform-independence, and scalability, while making a critical improvement of not interrupting throughput-oriented workloads. This is to be contrasted with the MVPP power capping design that uses Linux SIGSTOP and SIGKILL signals. This work also introduces the proactive capping sub-system to improve system availability, which the MVPP capping system does not have.

Our primary, reactive capping subsystem uses node-level CPU bandwidth control provided by the Linux kernel's CFS scheduler. To our knowledge this is the first time this node-level mechanism, applicable on a per-task basis, is used for data center power management. There is literature [2] that discusses using CPU bandwidth control for power management of mobile devices, but not for data centers. Other node-level mechanisms used for power management include DVFS [6, 16, 24], RAPL [25], Intel node manager [14], power gating [16], and thread packing [6, 17].

Reactive capping also uses load shaping, a data center-level closed-loop control, as the the power control policy. Load shaping is implemented at one level of the power delivery "choke point" that constrains the overall power capacity. It is

simpler than multi-level controls in other large-scale implementations [14, 24, 25].

Our failover, proactive capping subsystem to mitigate the risk of power signal unavailability, uses node-level CPU affinity control. It is the same low-level mechanism as "thread packing" [6, 17], but in this work we use it only as a failover mechanism when power signals are unavailable because of its limitations compared to CPU bandwidth control.

Other studies also use power-aware job scheduling and admission control to limit power draw [4, 12, 23]. Compared to node-level and hardware-level power throttling mechanisms such as ours, these scheduler-level techniques can improve availability and performance of running jobs. It is a valuable direction for future work, as discussed in Section 8.

10 Summary

In this paper we present Thunderbolt, a throughput-optimized and QoS-aware power capping system that is robust and scalable. We elaborate important design choices and present production evaluation of its policy decisions. Thunderbolt has been deployed in warehouse-sized data centers and saved us millions of dollars on capital expenses by enabling otherwise nonexistent additional power capacity in our data centers.

Acknowledgments

This work is the result of multi-year efforts contributed by many engineers, managers, and supporting staff. We would like to thank Strata Chalup, Greg Imwalle, Tom Kennedy, Dave Landhuis, Mian Luo, Matthew Nuckolls, Pablo Perez, Etienne Perot, Brad Strand, Jeff Swenson, Jikai Tang, Steve Webster, Quincy Ye, Henry Zhao, and Steven Zhao for their contributions and support for the Thunderbolt program. We are also grateful to David Culler, Gernot Heiser, Jeff Mogul, and the anonymous reviewers for their constructive feedback.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[2] Y. Ahn and K. Chung. User-centric power management for embedded CPUs using CPU bandwidth control. *IEEE Transactions on Mobile Computing*, 15(9):2388–2397, 2016.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[4] Arka A. Bhattacharya, David Culler, Aman Kansal, Sriram Govindan, and Sriram Sankar. The need for speed and stability in data center power capping. *Sustainable Computing: Informatics and Systems*, 2013.

[5] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1):1 – 14, 1989.

[6] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & cap: Adaptive DVFS and thread packing under power caps. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185, 2011.

[7] Intel Corporation. Power Stress and Shaping Tool. https://01.org/power-stress-and-shaping-tool. Accessed: 2020-04-15.

[8] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, 2010.

[9] Xing Fu, Xiaorui Wang, and Charles Lefurgy. How much power oversubscription is safe and allowed in data centers? In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 21–30, New York, NY, USA, 2011.

[10] Gartner, Inc. Gartner says global IT spending to decline 8% in 2020 due to impact of COVID-19. https://bit.ly/gartner-2020-05-13. Accessed: 2020-05-19.

[11] Synergy Research Group. Hyperscale operator spending on data centers up 11% in 2019 despite only modest capex growth. https://bit.ly/synergy-2020-03-24. Accessed: 2020-05-19.

[12] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 535–548. ACM, 2018.

[13] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *HPCA '08: 14th International Conference on High-Performance Computer Architecture*, pages 123–134, 2008.

[14] Y. Li, C. R. Lefurgy, K. Rajamani, M. S. Allen-Ware, G. J. Silva, D. D. Heimsoth, S. Ghose, and O. Mutlu. A scalable priority-aware approach to managing data center server power. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 701–714, 2019.

[15] Robert Love. CPU Affinity. https://www.linuxjournal.com/article/6799. Accessed: 2020-04-15.

[16] Kai Ma and Xiaorui Wang. PGCapping: exploiting power gating for power capping and core lifetime balancing in CMPs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[17] Sherief Reda, Ryan Cochran, and Ayse K. Coskun. Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 2012.

[18] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. Data center power oversubscription with a medium voltage power plane and priority-aware capping. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, pages 497–511. ACM, 2020.

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[20] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Paul Turner, Bharata B Rao, and Nikhil Rao. CPU bandwidth control for CFS. In *Proceedings of the Linux Symposium*, pages 245–254, 2010.

[22] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[23] Guosai Wang, Shuhao Wang, Bing Luo, Weisong Shi, Yinghang Zhu, Wenjun Yang, Dianming Hu, Longbo Huang, Xin Jin, and Wei Xu. Increasing large-scale data center capacity by statistical power control. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[24] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W. Keller. SHIP: A scalable hierarchical power control architecture for large-scale data centers. *IEEE Trans. Parallel Distrib. Syst.*, 23(1):168–176, 2012.

[25] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's data center-wide power management system. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 469–480, Piscataway, NJ, USA, 2016.

[26] Huazhe Zhang and Henry Hoffmann. A quantitative evaluation of the RAPL power control system. *Feedback Computing*, 2015.